

ThinkPHP3.1完全开发手册

哇塞悲哀



目 录

序言

1. 入门

- 1.1 简介
- 1.2 基础概念
- 1.3 获取ThinkPHP
- 1.4 环境要求
- 1.5 许可协议
- 1.6 目录结构
- 1.7 命名规范
- 1.8 MVC分层
- 1.9 CBD架构
- 1.10 特性概述
- 1.11 系统流程
- 1.12 开发流程

2. 入口

- 2.1 入口文件
- 2.2 项目目录
- 2.3 部署目录
- 2.4 项目编译
- 2.5 调试模式

3. 配置

- 3.1 配置格式
- 3.2 惯例配置
- 3.3 项目配置
- 3.4 调试配置
- 3.5 分组配置
- 3.6 读取配置
- 3.7 动态配置
- 3.8 扩展配置

4. 函数和类库

- 4.1 函数库
- 4.2 类库

5. 控制器

- 5.1 URL模式
- 5.2 模块和操作
- 5.3 定义控制器
- 5.4 空操作
- 5.5 空模块
- 5.6 模块分组

- 5.7 URL伪静态
- 5.8 URL路由
- 5.9 URL重写
- 5.10 URL生成
- 5.11 URL大小写
- 5.12 前置和后置操作
- 5.13 跨模块调用
- 5.14 页面跳转
- 5.15 重定向
- 5.16 获取系统变量
- 5.17 判断请求类型
- 5.18 获取URL参数
- 5.19 AJAX返回
- 5.20 Action参数绑定
- 5.21 多层控制器支持
- 6. 模型
 - 6.1 模型定义
 - 6.2 模型实例化
 - 6.3 字段定义
 - 6.4 数据主键
 - 6.5 属性访问
 - 6.6 跨库操作
 - 6.7 连接数据库
 - 6.8 切换数据库
 - 6.9 分布式数据库
 - 6.10 创建数据
 - 6.11 字段映射
 - 6.12 连贯操作
 - 6.13 CURD操作
 - 6.14 ActiveRecord
 - 6.15 自动验证
 - 6.16 命名范围
 - 6.17 自动完成
 - 6.18 查询语言
 - 6.19 查询锁定
 - 6.20 字段排除
 - 6.21 事务支持
 - 6.22 高级模型
 - 6.23 视图模型
 - 6.24 关联模型

- 6.25 Mongo模型
- 6.26 动态模型
- 6.27 虚拟模型
- 6.28 多层模型支持
- 7. 视图
 - 7.1 模板定义
 - 7.2 模板赋值
 - 7.3 模板输出
 - 7.4 模板替换
 - 7.5 获取内容
 - 7.6 模板引擎
 - 7.7 布局模板
- 8. 模板引擎
 - 8.1 变量输出
 - 8.2 系统变量
 - 8.3 使用函数
 - 8.4 默认值输出
 - 8.5 使用运算符
 - 8.6 内置标签
 - 8.7 包含文件
 - 8.8 导入文件
 - 8.9 Volist标签
 - 8.10 Foreach标签
 - 8.11 For标签
 - 8.12 Switch标签
 - 8.13 比较标签
 - 8.14 三元运算
 - 8.15 范围判断标签
 - 8.16 Present标签
 - 8.17 Empty标签
 - 8.18 Defined标签
 - 8.19 Define标签
 - 8.20 Assign标签
 - 8.21 IF标签
 - 8.22 标签嵌套
 - 8.23 使用PHP代码
 - 8.24 模板布局
 - 8.25 模板继承
 - 8.26 原样输出
 - 8.27 模板注释

- 8.28 引入标签库
 - 8.29 修改定界符
 - 8.30 避免JS混淆
- 9. 日志
 - 9.1 日志级别
 - 9.2 记录方式
 - 9.3 手动记录
- 10. 错误
 - 10.1 异常处理
 - 10.2 异常模板
 - 10.3 异常显示
- 11. 调试
 - 11.1 运行状态
 - 11.2 页面Trace
 - 11.3 调试方法
- 12. 缓存
 - 12.1 缓存方式
 - 12.2 动态缓存
 - 12.3 缓存队列
 - 12.4 快捷缓存
 - 12.5 快速缓存
 - 12.6 查询缓存
 - 12.7 SQL解析缓存
 - 12.8 静态缓存
- 13. 扩展
 - 13.1 行为扩展
 - 13.2 类库扩展
 - 13.3 控制器扩展
 - 13.4 模型扩展
 - 13.5 驱动扩展
 - 13.6 Widget扩展
 - 13.7 模式扩展
 - 13.8 引擎扩展
- 14. 安全
 - 14.1 表单令牌
 - 14.2 字段类型验证
 - 14.3 防止SQL注入
 - 14.4 输入过滤
 - 14.5 上传安全
 - 14.6 防止XSS攻击

- 14.7 其他安全建议
 - 14.8 目录安全文件
 - 14.9 保护模板文件
- 15. 性能
 - 15.1 关闭调试模式
 - 15.2 开启缓存
 - 15.3 合并字段缓存
 - 15.4 优化SQL
 - 15.5 替换入口
 - 15.6 前端优化
- 16. 部署
 - 16.1 PATH_INFO支持
 - 16.2 隐藏index.php
 - 16.3 二级域名部署
 - 16.4 定制错误页面
 - 16.5 设置时区
- 17. SAE支持
 - 17.1 SAE介绍
 - 17.2 获取SAE
 - 17.3 SAE开发
- 18. REST支持
 - 18.1 REST介绍
 - 18.2 REST模式
 - 18.3 REST配置
 - 18.4 REST路由
 - 18.5 REST方法
- 19. 杂项
 - 19.1 Session支持
 - 19.2 Cookie支持
 - 19.3 日期和时间
 - 19.4 WML开发
 - 19.5 多语言
 - 19.6 数据分页
 - 19.7 文件上传
 - 19.8 验证码
 - 19.9 图片添加水印
 - 19.10 IP获取和定位
 - 19.11Thinkphp的 I 方法
- 20. 附录
 - 20.1 常量参考

[20.2 配置参考](#)

[20.3 关于升级](#)

[20.4 大事记](#)

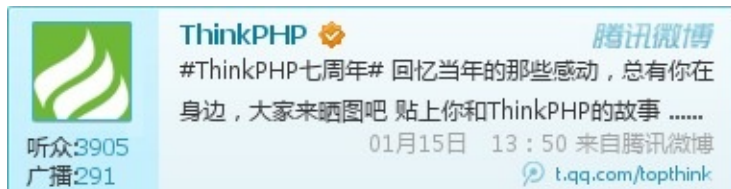
[鸣谢](#)

[关于](#)

序言

序言

[下一页](#)



版权申明

发布本资料须遵守开放出版许可协议 1.0 或者更新版本。

未经版权所有者明确授权，禁止发行本文档及其被实质上修改的版本。

未经版权所有者事先授权，禁止将此作品及其衍生作品以标准（纸质）书籍形式发行。

如果有兴趣再发行或再版本手册的全部或部分内容，不论修改过与否，或者有任何问题，请联系版权所有者 liu21st@gmail.com。

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

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

本文档的版权归ThinkPHP文档小组所有，本文档及其描述的内容受有关法律的版权保护，对本文档内容的任何形式的非法复制，泄露或散布，将导致相应的法律责任。

作者和贡献者

本手册内容由ThinkPHP创始人刘晨（流年）撰写，以及文档小组的成员参与贡献。

他们是misn、[麦当苗儿](#)、luofei、[deeka](#)、yangweijie、[vus520](#)。

并在此对所有参与手册纠错和建议的朋友表示感谢！

捐赠我们

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

我们的团队七年来一直在坚持不懈地努力，并坚持开源和免费提供使用，帮助开发人员更加方便的进行WEB应用的快速开发，如果您对我们的成果表示认同并且觉得对你有所帮助我们愿意接受来自各方面的捐赠^_^。



[下一页](#)

1. 入门

入门

[上一页](#)[下一页](#)

本章为您介绍学习ThinkPHP框架需要了解的基础概念和对ThinkPHP的概述介绍。

[上一页](#)[下一页](#)

1.1 简介

简介

[上一页](#)[下一页](#)

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应用开发而诞生的。ThinkPHP从诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简的代码的同时，也注重易用性。并且拥有众多的原创功能和特性，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。

经过6年的不断积累和重构，3.0版本又是一个新的里程碑版本，在框架底层的定制和扩展方面趋于完善，使得应用的开发范围和需求适应度更加扩大，能够满足不同程度的开发人员的需求。而且引入了全新的CBD（核心+行为+驱动）架构模式，旨在打造DIY框架和AOP编程体验，让ThinkPHP能够在不同方面都能快速满足项目和应用的需求，并且正式引入SAE、REST和Mongo支持。

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！经过6年来的不断重构和改进，ThinkPHP达到了一个新的阶段，能够满足企业开发中复杂的项目需求，足以达到企业级和门户级的开发标准。

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。

[上一页](#)[下一页](#)

1.2 基础概念

基础概念

[上一页](#)[下一页](#)

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。（以下基础概念的描述摘自互联网，仅供学习参考，更详细的说明请自行上网搜索）

LAMP

LAMP是基于Linux，Apache，MySQL和PHP的开放资源网络开发平台。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。

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。

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。

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。

AOP

AOP（Aspect-Oriented Programming，面向方面编程），可以说是OOP（Object-Oriented Programming，面向对象编程）的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。当我们需要为分散的对象引入公共行为的时候，OOP则显得无能为力。也就是说，OOP允许你定义从上到下的关系，但并不适合定义从左到右的关系。例如日志功能。日志代码往往水平地散布在所有对象层次中，而与它所散布到的对象的核心功能毫无关系。对于其他类型的代

码，如安全性、异常处理和透明的持续性也是如此。这种散布在各处的无关的代码被称为横切（cross-cutting）代码，在OOP设计中，它导致了大量代码的重复，而不利于各个模块的重用。而AOP技术则恰恰相反，它利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP代表的是一个横向的关系，如果说“对象”是一个空心的圆柱体，其中封装的是对象的属性和行为；那么面向方面编程的方法，就仿佛一把利刃，将这些空心圆柱体剖开，以获得其内部的消息。而剖开的切面，也就是所谓的“方面”了。然后它又以巧夺天工的妙手将这些剖开的切面复原，不留痕迹。

使用“横切”技术，AOP把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。正如Avanade公司的高级方案构架师Adam Magee所说，AOP的核心思想就是“将应用程序中的商业逻辑同对其提供支持的通用服务进行分离。”

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。

ActiveRecord

Active Record（中文名：活动记录）是一种领域模型模式，特点是一个模型类对应关系型数据库中的一个表，而模型类的一个实例对应表中的一行记录。Active Record 和 Row Gateway（行记录入口）十分相似，但前者是领域模型，后者是一种数据源模式。关系型数据库往往通过外键来表述实体关系，Active Record 在数据源层面上也将这种关系映射为对象的关联和聚集。Active Record 适合非常简单的领域需求，尤其在领域模型和数据库模型十分相似的情况下。如果遇到更加复杂的领域模型结构（例如用到继承、策略的领域模型），往往需要使用分离数据源的领域模型，结合 Data Mapper（数据映射器）使用。

Active Record 驱动框架一般兼有 ORM 框架的功能，但 Active Record 不是简单的 ORM，正如和 Row Gateway 的区别。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。

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。

[上一页](#)[下一页](#)

1.3 获取ThinkPHP

获取ThinkPHP

[上一页](#)[下一页](#)

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。
从3.1版本开始，官方仅发布核心框架，所有扩展和示例、文档均单独在官网和Github上面发布。

官网下载：

框架下载：<http://thinkphp.cn/down/framework.html>

扩展中心：<http://thinkphp.cn/extend.html>

示例中心：<http://thinkphp.cn/extend/example.html>

Github获取地址：

核心框架：<https://github.com/liu21st/thinkphp>

扩展中心：<https://github.com/liu21st/extend>

示例中心：<https://github.com/liu21st/examples>

原谷歌的SVN不再更新。

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

[上一页](#)[下一页](#)

1.4 环境要求

环境要求

[上一页](#)[下一页](#)

ThinkPHP3.0可以支持Windows/Unix服务器环境，需要PHP5.2.0以上版本支持，可运行于包括Apache、IIS和nginx在内的多种WEB服务器和模式，支持Mysql、MsSQL、PgSQL、Sqlite、Oracle、Ibase、Mongo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（[wampserver](#)是一个集成了Apache、PHP和MySQL的开发套件，而且支持不同PHP版本、MySQL版本和Apache版本的切换）来使用ThinkPHP进行本地开发和测试。

[上一页](#)[下一页](#)

1.5 许可协议

许可协议

[上一页](#)[下一页](#)

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>。

[上一页](#)[下一页](#)

1.6 目录结构

目录结构

[上一页](#)[下一页](#)

新版的目录结构在原来的基础上进行了调整，更加清晰。

| 目录/文件 | 说明 |

|-----|-----|

| ThinkPHP.php | 框架入口文件 |

| Common | 框架公共文件目录 |

| Conf | 框架配置文件目录 |

| Lang | 框架系统语言目录 |

| Lib | 系统核心基类库目录 |

| Tpl | 系统模板目录 |

| Extend | 框架扩展目录（关于扩展目录的详细信息请参考后面的扩展章节）|

注意：如果你下载的是核心版本，有可能Extend目录是空的，因为ThinkPHP本身不依赖任何扩展。

[上一页](#)[下一页](#)

1.7 命名规范

命名规范

[上一页](#)[下一页](#)

使用ThinkPHP开发的过程中应该尽量遵循下列命名规范：

- 类文件都是以.class.php为后缀（这里是指的ThinkPHP内部使用的类库文件，不代表外部加载的类库文件），使用驼峰法命名，并且首字母大写，例如DbMysql.class.php；
- 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，并且不同的类库的类命名有一定的规范；
- 函数、配置文件等其他类库文件之外的一般是以.php为后缀（第三方引入的不做要求）；
- 函数的命名使用小写字母和下划线的方式，例如 get_client_ip；
- 方法的命名使用驼峰法，并且首字母小写或者使用下划线“_”，例如 getUsername，_parseType，通常下划线开头的方法属于私有方法；
- 属性的命名使用驼峰法，并且首字母小写或者使用下划线“_”，例如 tableName、_instance，通常下划线开头的属性属于私有属性；
- 以双下划线“__”打头的函数或方法作为魔法方法，例如 call 和 __autoload；
- 常量以大写字母和下划线命名，例如 HAS_ONE和 MANY_TO_MANY；
- 配置参数以大写字母和下划线命名，例如HTML_CACHE_ON；
- 语言变量以大写字母和下划线命名，例如MY_LANG，以下划线打头的语言变量通常用于系统语言变量，例如 _CLASS_NOTEXIST；
- 对变量的命名没有强制的规范，可以根据团队规范来进行；
- ThinkPHP的模板文件默认是以.html 为后缀（可以通过配置修改）；
- 数据表和字段采用小写加下划线方式命名，并注意字段名不要以下划线开头，例如 think_user 表和 user_name字段，类似 _username 这样的数据表字段可能会被过滤。

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

另外有一点非常关键，ThinkPHP默认全部使用UTF-8编码，所以请确保你的程序文件采用UTF-8编码格式保存，并且去掉BOM信息头（去掉BOM头信息有很多方式，不同的编辑器都有设置方法，也可以用工具进行统一检测和处理），否则可能导致很多意想不到的问题。

[上一页](#)[下一页](#)

1.8 MVC分层

MVC分层

[上一页](#)[下一页](#)

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。

但实际上，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。

[上一页](#)[下一页](#)

1.9 CBD架构

CBD架构

[上一页](#)[下一页](#)

ThinkPHP3.0版本引入了全新的CBD（核心Core+行为Behavior+驱动Driver）架构模式，因为从底层开始，框架就采用核心+行为+驱动的架构体系，核心保留了最关键的部分，并在重要位置设置了标签用以标记，其他功能都采用行为扩展和驱动的方式组合，开发人员可以根据自己的需要，对某个标签位置进行行为扩展或者替换，就可以方便的定制框架底层，也可以在应用层添加自己的标签位置和添加应用行。而标签位置类似于AOP概念中的“切面”，行为都是围绕这个“切面”来进行编程，如果把系统内置的核心扩展看成是一种标准模式的话，那么用户可以把这一切的行为定制打包成一个新的模式，所以在ThinkPHP里面，称之为模式扩展，事实上，模式扩展不仅仅可以替换和增加行为，还可以对底层的MVC进行替换和修改，以达到量身定制的目的。利用这一新的特性，开发人员可以方便地通过模式扩展为自己量身定制一套属于自己或者企业的开发框架，新版的模式扩展是框架扩展的集大成者，开创了新的里程碑，这正是新版的真正魅力所在。

[上一页](#)[下一页](#)

1.10 特性概述

特性概述

[上一页](#) [下一页](#)

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、模板引擎、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、多数据库支持、认证机制和扩展性方面均有独特的表现。

值得推荐的特性包括：

CBD架构：ThinkPHP3.0版本引入了全新的CBD（核心+行为+驱动）架构模式，打造框架底层DIY定制和类AOP编程体验。利用这一新的特性，开发人员可以方便地通过模式扩展为自己量身定制一套属于自己或者企业的开发框架。

编译机制：独创的项目编译机制，有效减少OOP开发中文件加载的性能开销。改进后的项目编译机制，可以支持编译文件直接作为入口载入，并且支持常量外部载入，利于产品发布。

类库导入：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，而且还支持自动加载和别名导入。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和路由：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。支持灵活的规则路由和正则路由，以及路由重定向支持，带给开发人员更方便灵活的URL优化体验。

调试模式：框架提供的调试模式可以方便用于开发过程的不同阶段，包括开发、测试和演示等任何需要的情况，不同的应用模式可以配置独立的项目配置文件。只是小小的性能牺牲就能满足调试开发过程中的日志和分析需要，并确保将来的部署顺利，一旦切换到部署模式则可以迅速提升性能。

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。

数据库：支持包括Mysql、Sqlite、Pgsq、Oracle、SqlServer、Mongo等数据库，并且内置分布式数据库和读写分离功能支持。系统支持多数据库连接和动态切换机制，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。

查询语言：内建丰富的查询机制，包括组合查询、快捷查询、复合查询、区间查询、统计查询、定位查询、多表查询、子查询、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。

扩展模型：提供了丰富的扩展模型，包括：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的高级模型；可以轻松动态地创建数据库视图的视图模型；支持关联操作的关联模型；支持Mongo数据库的Mongo模型等等，都可以方便的使用。

模块分组：不用担心大项目的分工协调和部署问题，分组帮你解决跨项目的难题，还可以支持对分组的二级域名部署支持。

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，并内置布局模板功能和标签库扩展支持。通过驱动还可以支持Smarty、EaseTemplate、TemplateLite、Smart等其他第三方模板引擎。

AJAX支持：内置和客户端无关的AJAX数据返回方法，支持JSON、XML和EVAL类型返回客户端，而且可以扩展返回数据格式，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。

SAE支持：提供了新浪SAE平台的强力支持，具备“横跨性”和“平滑性”，支持本地化开发和调试以及部署切换，让你轻松过渡到SAE开发，打造全新的SAE开发体验。

RESTFul支持：REST模式提供了RESTFul支持，为你打造全新的URL设计和访问体验，同时为接口应用提供了支持。

多语言支持：系统支持语言包功能，项目和分组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，还提供了AMF、PHPRpc、Lite、Thin和Cli模式扩展支持，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，新版新增了IP验证和有效期验证等更多的验证方式，配合自动完成可以生成安全的数据对象。

字段类型检测：系统会自动缓存字段信息和字段类型，支持非法字段过滤和字段类型强制转换，确保数据写入和查询更安全。

缓存机制：系统支持包括文件方式、APC、Db、Memcache、Shmop、Sqlite、Redis、Eaccelerator和Xcache在内的动态数据缓存类型，以及可定制的静态缓存规则，并提供了快捷方法进行存取操作。

扩展机制：系统支持包括模式扩展、行为扩展、类库扩展、驱动扩展、模型扩展、控制器扩展、Widget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，满足企业开发中更加复杂的项目需求。

[上一页](#)[下一页](#)

1.11 系统流程

系统流程

[上一页](#)[下一页](#)

我们以访问网址 <http://serverName.com/index.php/User/read/id/8> 为例，分两种情况来解析下系统的执行流程，首先是调试模式下面的主要执行流程：

序号	流程说明
1	用户访问网站URL地址
2	调用项目的入口文件（这里是index.php）
3	载入系统入口文件ThinkPHP.php
4	判断系统常量，如果没有定义则自动生成
5	载入系统运行时文件runtime.php并定义项目路径常量
6	加载运行时所需的文件（通过调用load_runtime_file函数）
7	后面的流程和调试模式基本相同，只是模板编译过程省略了
8	读取核心基础文件列表和加载系统别名定义文件
9	检查项目相关目录是否存在，不存在则自动生成
10	调用Think::start执行入口
11	设置异常和错误处理机制
12	注册系统自动加载机制
13	预编译当前项目
14	加载框架惯例配置文件
15	读取当前的运行模式如果不是标准模式则加载模式的配置文件（如果存在）
16	加载模式和项目配置文件
17	加载框架底层语言包文件
18	加载当前模式的系统行为定义文件
19	加载当前模式的项目行为定义文件（如果存在）
20	读取核心编译文件列表
21	载入项目公共函数文件
22	加载模式和项目别名定义文件
23	加载系统调试模式配置文件
24	加载项目调试模式配置文件（如果存在）

25	执行当前模式的App::run();运行项目
26	如果定义了动态载入则载入动态项目配置文件和公共文件
27	URL调度，根据URL模式设置分析当前URL地址
28	URL路由检测
29	获取当前URL地址的分组、模块和操作名 及其他参数并生成URL相关常量定义
30	如果检测到分组，则加载分组的配置文件和公共文件
31	检测模板主题并生成模板系统常量
32	设置SESSION_ID 开启Session
33	根据分组和模块名，定位到控制器类并且实例化
34	检查并执行当前操作的前置方法
35	检查当前模块的_initialize方法
36	执行当前操作方法
37	调用控制器的Display方法输出
38	定位当前操作方法的模板文件
39	调用模板引擎解析模板内容并生成模板编译缓存文件
40	读取模板缓存文件进行变量输出，替换解析返回的内容中的需要替换的特殊字符串
41	生成表单令牌哈希
42	输出模板内容到浏览器
43	如果开启页面Trace显示则调用trace信息显示
44	检查并执行当前操作的后置方法
45	项目运行结束，记录内存中的日志信息到文件

如果在部署模式下面（假设已经生成项目编译缓存），基本的系统流程是：

| 序号 | 流程说明 |

|-----|-----|

| 1 | 用户访问网站URL地址 |

| 2 | 调用项目的入口文件，如果替换了入口文件，则调用项目编译缓存文件，并跳过下面的3、4、5流程，直接执行后面的流程。 |

| 3 | 载入系统入口文件ThinkPHP.php |

| 4 | 判断系统常量，如果没有定义则自动生成 |

| 5 | 载入系统运行时文件runtime.php并定义项目路径常量 |

| 6 | 加载运行时所需的文件（通过调用load_runtime_file函数） |

| 7 | 加载系统基础函数库文件common.php |

系统执行流程根据不同的设置、行为和模式影响，可能存在差异，并不一定完整。但是开启页面Trace功能后，你就可以比较直观的看到当前的文件载入流程，能够帮助你了解系统的执行流程，例如新版的blog

示例在关闭调试模式下面一共加载了20个文件，列表如下：

```
[0] => E:\www\App\Examples\Blog\index.php
[1] => E:\www\App\ThinkPHP\ThinkPHP.php
[2] => E:\www\App\Examples\Blog\Runtime\~runtime.php
[3] => E:\www\App\Examples\Blog\Lib\Behavior\CheckLangBehavior.class.php
[4] => E:\www\App\Examples\Blog\Lib\Action\BlogAction.class.php
[5] => E:\www\App\Examples\Blog\Lib\Action\PublicAction.class.php
[6] => E:\www\App\ThinkPHP\Lib\Core\Model.class.php
[7] => E:\www\App\ThinkPHP\Lib\Core\Db.class.php
[8] => E:\www\App\ThinkPHP\Lib\Driver\Db\DbMysql.class.php
[9] => E:\www\App\Examples\Blog\Runtime\Data\_fields\examples.Category.php
[10] => E:\www\App\Examples\Blog\Lib\Model\AdvModel.class.php
[11] => E:\www\App\Examples\Blog\Runtime\Data\_fields\examples.Blog.php
[12] => E:\www\App\Examples\Blog\Runtime\Data\_fields\examples.Comment.php
[13] => E:\www\App\Examples\Blog\Runtime\Data\_fields\examples.Tag.php
[14] => E:\www\App\Examples\Blog\Lib\Model\BlogViewModel.class.php
[15] => E:\www\App\Examples\Blog\Lib\Model\ViewModel.class.php
[16] => E:\www\App\Examples\Blog\Lib\Model\BlogModel.class.php
[17] => E:\www\App\Examples\Blog\Lib\Model\CategoryModel.class.php
[18] => E:\www\App\Examples\Blog\Lib\ORG\Page.class.php
[19] => E:\www\App\Examples\Blog\Runtime\Cache\2ab73b774a28fab5232b8c752b6540
```

[上一页](#)[下一页](#)

1.12 开发流程

开发流程

[上一页](#)[下一页](#)

使用ThinkPHP创建应用的一般开发流程是：

- 系统设计、创建数据库和数据表；（可选）
- 项目命名并创建项目入口文件，开启调试模式；
- 完成项目配置；
- 创建项目函数库；（可选）
- 开发项目需要的扩展（模式、驱动、标签库等）；（可选）
- 创建控制器类；
- 创建模型类；（可选）
- 创建模板文件；
- 运行和调试、分析日志；
- 开发和设置缓存功能；（可选）
- 添加路由支持；（可选）
- 安全检查；（可选）
- 部署到生产环境。

下面我们会详细描述如何在不同的环节使用ThinkPHP来最大程度地简化开发，体验使用ThinkPHP开发的乐趣。

[上一页](#)[下一页](#)

2. 入口

入口

[上一页](#)[下一页](#)

本章为您介绍如何创建的项目入口文件。

[上一页](#)[下一页](#)

2.1 入口文件

入口文件

[上一页](#) [下一页](#)

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，一个项目都有一个统一（但不一定是唯一）的入口。应该说，所有项目都是从入口文件开始的，并且所有的项目的入口文件是类似的，入口文件中主要包括：

- 定义框架路径、项目路径和项目名称（可选）
- 定义调试模式和运行模式的相关常量（可选）
- 载入框架入口文件（必须）

首先，在服务器或者本地的Web目录下面创建一个App目录，并且把下载的ThinkPHP框架的ThinkPHP目录拷贝到App目录下面，然后在App目录下面创建一个index.php文件，该文件就是我们要创建项目的入口文件。


新版的入口文件更加简化，默认情况下，只需要在该文件中添加一行代码即可：

```
<?php
//加载框架入口文件
require './ThinkPHP/ThinkPHP.php';
```

然后，我们打开浏览器，输入地址并运行：

<http://localhost/App/>

就会看到欢迎页面：



表示ThinkPHP已经成功执行，这个时候，系统已经在App下面自动生成了项目相关目录，并写入了初始Action。（注意：如果是类Unix或者Linux环境下测试的话，需要对App目录设置可写权限，否则无法自动生成目录结构）入口文件中还可以添加系统或者应用的常量定义，如果我们的项目需要采用其他的模式运行（例如，采用命令行模式运行），那么可以定义MODE_NAME如下：

```
define('MODE_NAME', 'cli');
```

如果没有在项目入口文件中设置MODE_NAME常量的话，就表示采用系统的标准模式运行。由于模式扩展可以改变底层的运行机制和行为定义，本手册中的内容如无特别说明，功能描述均表示运行于标准模式下面。

入口文件并不一定都是指index.php文件，因为我们可以为不同的项目创建不同的入口文件，例如，前台项目的入口文件为index.php，后台项目的入口文件可能是admin.php。

[上一页](#) [下一页](#)

2.2 项目目录

项目目录

[上一页](#)[下一页](#)

生成的项目目录结构和系统目录类似，包括：

| 目录 | 说明 |

|-----|-----|

| 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，项目所有的配置文件都放在这里 |

| Lang | 项目语言包目录（可选 如果不需要多语言支持 可删除） |

| 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，如果存在分组的话，则首先是分组目录。 |

如果需要把index.php 移动到App目录的外面，只需要在入口文件中增加项目名称和项目路径定义。

```
<code><?php
//定义项目名称
define('APP_NAME', 'App');
//定义项目路径
define('APP_PATH', './App/');
//加载框架入文件
require './App/ThinkPHP/ThinkPHP.php'; APP_NAME 是指项目名称，注意APP_NAME

```

不要随意设置，通常是项目的目录名称，如果你的项目是直接部署在Web根目录下面的话，那么需要设置APP_NAME 为空。

APP_PATH 是指项目路径（必须以“/”结束），项目路径是指项目的Common、Lib目录所在的位置，而不是项目入口文件所在的位置。

注意：在类Unix或者Linux环境下面Runtime目录需要可写权限。

[上一页](#)[下一页](#)

2.3 部署目录

部署目录

[上一页](#) [下一页](#)

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：

目录/文件	说明
ThinkPHP	系统目录（下面的目录结构同上面的系统目录）
Public	网站公共资源目录（存放网站的Css、Js和图片等资源）
Uploads	网站上传目录（用户上传的统一目录）
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	项目Home的入口文件
admin.php	项目Admin的入口文件
..... 更多的项目入口文件	

如果采用分组模块的话 可以简化为一个项目目录

目录/文件	说明
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（分组目录结构会在后面描述）
Public	网站公共资源目录（存放网站的Css、Js和图片等资源）
Uploads	网站上传目录（用户上传的统一目录）
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。

事实上，系统目录和项目目录可以放到非WEB访问目录下面，网站目录下面只需要放置Public公共目录和入口文件，从而提高网站的安全性。

[上一页](#) [下一页](#)

本文档使用 [看云](#) 构建

2.4 项目编译

项目编译

[上一页](#)[下一页](#)

项目编译机制作为ThinkPHP独创的功能特色，从1.0版本就延续至今，编译缓存的基础原理是第一次运行的时候把核心需要加载的文件去掉空白和注释后合并到一个文件中，第二次运行的时候就直接载入编译缓存而无需载入众多的核心文件，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。3.0版本的项目编译更是带来了新的飞跃，包括：

- 首先是合并了2.0体系的核心编译缓存和项目编译缓存，不再生成两个缓存文件；
- 其次是融合了之前ALLINONE模式，直接对本地环境生成设置和常量定义，减少环境判断有效提升性能；
- 更具特色的是新版的编译缓存可以直接替换框架入口甚至网站入口，从某种程度来说，编译后的框架甚至可以脱离框架核心独立运行；
- 还可以通过参数设置，生成的编译缓存载入外部的常量定义文件，便于产品做用户定义；

因为刚才我们并没有开启调试模式，所以第一次运行之后，除了已经自动生成目录结构外，同时也已经生成了编译缓存文件了。

编译缓存文件默认生成在项目的Runtime目录下面，我们可以在App/Runtime目录下面看到有一个~runtime.php文件，这个就是编译缓存文件。

如果你使用了模式扩展的话，编译缓存文件名称可能会有所变化，例如，如果你当前用的是REST模式，那么生成的编译缓存文件则会变成~rest_runtime.php。

注意：环境改变后需要删除编译缓存文件，也就是说你不能把本地生成的编译缓存拷贝到服务器或者其他环境直接使用。

编译缓存的内容通常包括：系统函数库、系统基础核心类库、核心或者扩展定义的核心行为类库、项目配置文件、项目函数文件。如果希望自己设置目录，可以在入口文件里面更改RUNTIME_PATH常量进行更改，例如：`define('RUNTIME_PATH', './App/temp/')`；注意RUNTIME_PATH目录必须设置为可写权限。

除了自定义编译缓存目录之外，还支持自定义编译缓存文件名，例如：

`define('RUNTIME_FILE', './App/temp/runtime_cache.php')`；接下来要展示一个新版编译缓存的新特性，假如我们之前已经生成了App/Runtime/~runtime.php编译缓存文件，现在我们进行入口

```
<?php
// 替换入口文件为编译缓存文件
```

文件替换，修改入口文件如下：`require './App/Runtime/~runtime.php'`；再次执行后运行依然正常，这个时候其实入口已经被编译缓存文件接管了，跳过了框架的入口文件

本文档使用 [看云](#) 构建

ThinkPHP/ThinkPHP.php。

接下来，见证奇迹的时刻到来了^_^，我们把项目的入口文件index.php删除，并且把编译缓存文件拷贝到项目目录下面，更名为index.php，再次执行运行正常，说明我们已经跳过了入口文件，直接以编译缓存文件为项目运行入口了。

[上一页](#)[下一页](#)

2.5 调试模式

调试模式

[上一页](#)[下一页](#)

虽然编译缓存很优秀，但是并不利于开发阶段中调试和排错，我们强烈建议ThinkPHP开发人员在开发阶段始终开启调试模式，方便及时发现隐患问题和分析、解决问题。开启调试模式很简单，只需要在入口文

```
<?php
//开启调试模式
define('APP_DEBUG', true);
//加载框架入口文件
```

件中增加一行常量定义代码：`require './ThinkPHP/ThinkPHP.php'`；在完成开发阶段部署到生产环境后，只需要删除调试模式定义代码即可切换到部署模式。开启调试模式后，系统会首先加载系统默认的调试配置文件，然后加载项目的调试配置文件，调试模式的优势在于：

- 开启日志记录，任何错误信息和调试信息都会详细记录，便于调试；
- 关闭模板缓存，模板修改可以即时生效；
- 记录SQL日志，方便分析SQL；
- 关闭字段缓存，数据表字段修改不受缓存影响；
- 严格检查文件大小写（即使是Windows平台），帮助你提前发现Linux部署问题；
- 可以方便用于开发过程的不同阶段，包括开发、测试和演示等任何需要的情况，不同的应用模式可以配置独立的项目配置文件；

关于调试模式的更多用法，我们会在后面进行更详细的讲解。

[上一页](#)[下一页](#)

3. 配置

配置

[上一页](#)[下一页](#)

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、分组配置、调试配置和动态配置，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，你还可以增加动态配置文件。

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->扩展配置->动态配置

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

[上一页](#)[下一页](#)

3.1 配置格式

配置格式

[上一页](#)[下一页](#)

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：

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

配置参数不区分大小写（因为无

论大小写定义都会转换成小写），所以下面的配置等效：

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

但是我们建议保持大写定义配置

参数的规范。还可以在配置文件中可以使用二维数组来配置更多的信息，例如：

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

需要注意的是，二级参数配置区

分大小写，也就说读取确保和定义一致。

[上一页](#)[下一页](#)

3.2 惯例配置

惯例配置

[上一页](#)[下一页](#)

惯例重于配置是系统遵循的一个重要思想，系统内置有一个惯例配置文件（位于系统目录下面的 `Conf\convention.php`），按照大多数的使用对常用参数进行了默认配置。所以，对于应用项目的配置文件，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。

惯例配置文件会被系统自动加载，无需在项目中进行加载。

注意：因为新版系统架构的改变，部分惯例配置参数已经分离出来纳入了行为扩展的属性参数（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）。

[上一页](#)[下一页](#)

3.3 项目配置

项目配置

[上一页](#)[下一页](#)

项目配置文件是最常用的配置文件，项目配置文件位于项目的配置文件目录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。

[上一页](#)[下一页](#)

3.4 调试配置

调试配置

[上一页](#)[下一页](#)

新版增强了调试模式的配置文件，在开启调试模式的状态下，可以给项目设置不同的应用状态，并加载不同的项目配置文件，但是无论如何，都会首先导入框架默认的调试模式配置文件，该文件位于系统目录的 `Conf\debug.php`。

通常情况下，调试配置文件里面可以进行一些开发模式所需要的配置。例如，配置额外的数据库连接用于调试，开启日志写入便于查找错误信息、开启页面Trace输出更多的调试信息等等。

注意：3.0版本的调试模式默认没有开启运行时间显示和页面Trace显示，需要自行开启，并且建议调试模式只开启页面Trace即可，新版的页面Trace显示信息已经包含了运行时间显示。

如果没有配置应用状态，系统默认则默认为debug状态，也就是说默认的配置参数是：

```
'APP_STATUS' => 'debug', //应用调试模式状态 如果检测到项目的配置目录中有存在debug.php文件，则会自动加载该配置文件，并且和系统项目配置文件以及系统调试配置文件合并，也就是说，debug.php配置文件只需要配置和项目配置文件以及系统调试配置文件不同的参数或者新增的参数。
```

如果想在调试模式下面增加应用状态，例如测试状态，则可以在项目配置文件中改变设置如下：

```
'APP_STATUS' => 'test', //应用调试模式状态 这样的话，系统会自动尝试加载项目配置目录下面的test.php 配置文件，可以在test配置文件中改变相关设置，例如改变测试数据库的连接信息等等。
```

由于调试模式没有任何缓存，因此涉及到较多的文件IO操作和模板实时编译，所以在开启调试模式的情况下，性能会有一定的下降，但不会影响部署模式的性能。

注意：一旦关闭调试模式，项目的调试配置文件即刻失效。

[上一页](#)[下一页](#)

3.5 分组配置

分组配置

[上一页](#)[下一页](#)

如果启用了模块分组，则可以在对每个分组单独定义配置文件，分组配置文件位于：

项目配置目录/分组名称/config.php

可以通过如下配置启用分组：
`'APP_GROUP_LIST' => 'Home,Admin', //项目分组设定`
`'DEFAULT_GROUP' => 'Home', //默认分组` 现在定义了Home和Admin两个分组，则我们可以定义分组配置文件如下：

Conf/Home/config.php

Conf/Admin/config.php

每个分组的配置文件仅在当前分组有效，分组配置的定义格式和项目配置是一样的。

注意：分组名称区分大小写，必须和定义的分组名一致。

[上一页](#)[下一页](#)

3.6 读取配置

读取配置

[上一页](#)[下一页](#)

定义了配置文件之后，可以使用系统提供的C方法（如果觉得比较奇怪的话，可以借助Config单词来帮助记忆）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_STATUS')` 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_status')`是等效的，但是建议使用大写方式的规范。

如果APP_STATUS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')` // 获取用户配置中的用户类型设置 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。

[上一页](#)[下一页](#)

3.7 动态配置

动态配置

[上一页](#)[下一页](#)

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（类似的双关用法在ThinkPHP的系统设计中较为常见）。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')`；3.1版本开始，C函数支持配置保存功能，仅对批量设置有效，使用方法：`C($array, 'name')`；其中array是一个数组变量，会把批量设置后的配置参数列表保存到name标识的缓存数据中

获取缓存的设置列表数据 可以用 `C('', 'name')`；//或者`C(null, 'name')`；会读取name标识的缓存配置数据到当前配置数据（合并）。

[上一页](#)[下一页](#)

3.8 扩展配置

扩展配置

[上一页](#)[下一页](#)

项目配置文件在部署模式的时候会纳入编译缓存，也就是说编译后再修改项目配置文件就不会立刻生效，需要删除编译缓存后才能生效。扩展配置文件则不受此限制影响，即使在部署模式下面，修改配置后可以实时生效，并且配置格式和项目配置一样。

设置扩展配置的方式如下（多个文件用逗号分隔）：

`'LOAD_EXT_CONFIG' => 'user,db',` // 加载扩展配置文件 项目设置了加载扩展配置文件 `user.php` 和 `db.php` 分别用于用户配置和数据库配置，那么会自动加载项目配置目录下面的配置文件 `Conf/user.php` 和 `Conf/db.php`。

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

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

`user.php` 中的配置参数如下：那么，最终

获取用户参数的方式是：`C('USER_AUTH_ID')`；如果希望采用二级配置方式，可以设置如下：

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

同样的`user.php` 配置文件内容，但最终获取用户参数的方式就变成了：

`C('USER.USER_AUTH_ID')`；这种方式可以避免大项目情况中的参数冲突问题。下面的一些配置文件已经被系统使用，请不要作为自定义的扩展配置重新定义：

文件名	说明
config.php	项目配置文件
tags.php	项目行为配置文件
alias.php	项目别名定义文件
debug.php	项目调试模式配置文件（以及项目设置的APP_STATUS对应的配置文件）
core.php	项目追加的核心编译列表文件（不会覆盖核心编译列表）

[上一页](#)[下一页](#)

4. 函数和类库

函数和类库

[上一页](#)[下一页](#)

本章为您介绍下ThinkPHP的函数和类库的基础用法。

[上一页](#)[下一页](#)

4.1 函数库

函数库

[上一页](#) [下一页](#)

ThinkPHP中的函数库可以分为系统函数库和项目函数库。

系统函数库

库系统函数库位于系统的Common目录下面，有三个文件：

common.php是全局必须加载的基础函数库，在任何时候都可以直接调用；

functions.php是框架标准模式的公共函数库，其他模式可以替换加载自己的公共函数库或者对公共函数库中的函数进行重新定义；

runtime.php是框架运行时文件，仅在调试模式或者编译过程才会被加载，因此其中的方法在项目中不能直接调用；

项目函数库

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。

如果项目配置中使用了动态函数加载配置的话，项目Common目录下面可能会存在更多的函数文件，动态加载的函数文件不会纳入编译缓存。

在特殊的情况下，模式可以改变自动加载的项目函数库的位置或者名称。

扩展函数库

库我们可以在项目公共目录下面定义扩展函数库，方便需要的时候加载和调用。扩展函数库的函数定义规范和项目函数库一致，只是函数库文件名可以随意命名，一般来说，扩展函数库并不会自动加载，除非你设置了动态载入。

函数加载

系统函数库和项目函数库中的函数无需加载就可以直接调用，对于项目的扩展函数库，可以采用下面两种方式调用：

动态载入

我们可以在项目配置文件中定义LOAD_EXT_FILE参数，例如：`"LOAD_EXT_FILE"=>"user,db"` 通过上面的设置，就会执行过程中自动载入项目公共目录下面的扩展函数库文件user.php和db.php，这样就可以直接在项目中调用扩展函数库user.php和db.php中的函数了，而且扩展函数库的函数修改是实时生效的。

手动载入

如果你的函数只是个别模块偶尔使用，则不需要采用自动加载方式，可以在需要调用的时候采用load方法手动载入，方式如下：`load("@.user")` @.user表示加载当前项目的user函数文件，这样就可以直接user.php扩展函数库中的函数了。

[上一页](#)[下一页](#)

4.2 类库

类库

[上一页](#)[下一页](#)

ThinkPHP的类库包括基类库和应用类库，系统的类库命名规则如下：

类库	规则	示例
控制器类	模块名+Action	例如 UserAction、InfoAction
模型类	模型名+Model	例如 UserModel、InfoModel
行为类	行为名+Behavior	例如 CheckRouteBehavior
Widget类	Widget名+Widget	例如 BlogInfoWidget
驱动类	引擎名+驱动名	例如 DbMysql表示mysql数据库驱动、CacheFile表示文件缓存驱动

类名和文件名一致，详细命名规范可以参考1.6 命名规范。

基类库

基类库是指符合ThinkPHP类库规范的系统类库，包括ThinkPHP的核心基类库和扩展基类库。核心基类库目录位于系统的Lib目录，核心基类库也就是Think类库，扩展基类库位于Extend/Library目录，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内置支持类等，包含有：

目录	调用路径	说明
Lib/Core	Think.Core	核心类库包
Lib/Behavior	Think.Behavior	内置行为类库包
Lib/Driver	Think.Driver	内置驱动类库包
Lib/Template	Think.Template	内置模板引擎类库包

核心类库包下面包含下面核心类库：

类名	说明
Action	系统基础控制器类
App	系统应用类
Behavior	系统行为基础类
Cache	系统缓存类
Db	系统抽象数据库类
Dispatcher	URL调度类
Log	系统日志类
Model	系统基础模型类
Think	系统入口和静态类

本文档使用 [看云](#) 构建

ThinkException	系统基础异常类
View	视图类
Widget	系统Widget基础类

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，通常包括：

目录	调用路径	说明
Lib/Action	@.Action或自动加载	控制器类库包
Lib/Model	@.Model或自动加载	模型类库包
Lib/Behavior	用B方法调用或自动加载	应用行为类库包
Lib/Widget	用W方法在模板中调用	应用Widget类库包

项目根据自己的需要可以在项目类库目录下面添加自己的类库包，例如Lib/Common、Lib/Tool等。

类库导入

ThinkPHP类库的导入区别于其他的框架并没有采用require或者require_once进行导入，所有类库导入都采用统一的机制，包含下面两种方式：

一、Import显式导入

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，如果导入了不同的位置下面的同名类库文件，系统也不会再次导入，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，所以系统不会再次导入ORG.Util.Array类。

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

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包的导入作为基类库导入，否则就认为是项目应用类库导入。`import("Think.Util.Session");`
`import("ORG.Util.Page");` 上面两个方法分别导入了Think基类库的Util/Session.class.php文件和ORG扩展类库包的Util/Page.class.php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__));
```

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

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

我们建议您使用ThinkPHP开发过程保持类库名称采用class.php的后缀规范。

二，别名导入

除了命名空间的导入方式外，import方法还可以支持别名导入，要使用别名导入，首先要定义别名，我们可以在项目配置目录下面增加alias.php用以定义项目中需要用到的类库别名，例如：

```
return array(
    'rbac' =>LIB_PATH.'Common/Rbac.class.php',
    'page' =>LIB_PATH.'Common/Page.class.php',
);
```

那么，现在就可以直接使用：

```
import("rbac");
```

```
import("page");
```

导入Rbac和Page类，别名导入方式禁止使用import方法的第二和第三个参数，别名导入方式的效率比命名空间导入方式要高效，缺点是需要预先定义相关别名。

可以为某些需要的类库定义别名，那么无需定义自动加载路径也可以快速的自动加载。

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扩展目录下的Vendor目录，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。

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

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

自动加载

在大多数情况下，我们无需手动导入类库，而是通过配置采用自动加载机制即可，自动加载机制是真正的

本文档使用 [看云](#) 构建

按需加载，可以很大程度的提高性能。自动加载有三种情况，按照加载优先级从高到低分别是：别名自动加载、系统规则自动加载和自定义路径自动加载。

一、别名自动加载

在前面我们提到了别名的定义方式，并且采用了import方法进行别名导入，其实所有定义别名的类库都无需再手动加载，系统会按需自动加载。

二、系统规则自动加载

果你没有定义别名的话，系统会首先按照内置的规则来判断加载，系统规则仅针对行为类、模型类和控制器类，搜索规则如下：

类名	规则	说明
行为类	规则1	搜索系统类库目录下面的Behavior目录
	规则2	搜索系统扩展目录下面的Behavior目录
	规则3	搜索应用类库目录下面的Behavior目录
	规则4	如果启用了模式扩展，则搜索模式扩展目录下面的Behavior目录
模型类	规则1	如果启用分组，则搜索应用类库目录的Model/当前分组 目录
	规则2	搜索应用类库下面的Model目录
	规则3	搜索系统扩展目录下面的Model目录
控制器类	规则1	如果启用分组，则搜索应用类库目录的Action/当前分组 目录
	规则2	搜索项目类库目录下面的Action目录
	规则3	搜索系统扩展目录下面的Action目录

注意：搜索的优先顺序从上至下，一旦找到则返回，后面规则不再检测。如果全部规则检测完成后依然没有找到类库，则开始进行第三个自定义路径自动加载检测。

三、自定义路径自动加载

当你的类库比较集中在某个目录下面，而且不想定义太多的别名导入的话，可以使用自定义路径自动加载方式，这种方式需要在项目配置文件中添加自动加载的搜索路径，例如：

`'APP_AUTOLOAD_PATH' => '@.Common,@.Tool'`，表示，在当前项目类库目录下面的Common和Tool目录下面的类库可以自动加载。多个搜索路径之间用逗号分割，并且注意定义的顺序也就是自动搜索的顺序。

注意：自动搜索路径定义只能采用命名空间方式，也就是说这种方式只能自动加载项目类库目录和基类库目录下面的类库文件。

[上一页](#) [下一页](#)

5. 控制器

控制器

[上一页](#)[下一页](#)

ThinkPHP的控制器就是Action类，如何设计控制器取决于你的URL的规划。控制器和模型并没有直接的关联，你可以在一个控制器里面操作任何的模型。

[上一页](#)[下一页](#)

5.1 URL模式

URL模式

[上一页](#) [下一页](#)

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` 二、PATHINFO模式（默认模式）：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/` 在不考虑路由的情况下，第一个参数会被解析成模块名称（如果启用了分组的话，则依次往后递推），第二个参数会被解析成操作，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，还可以和URL_WRITE模式整合。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，但是在大多数环境下面ThinkPHP可以进行兼容判断，如果你的服务器环境或者空间仍然无法识别PATHINFO的话，或者需要自己增加识别方法或者可以选择普通模式或者兼容模式URL运行。我们建议的方式是采用PATHINFO模式开发，如果部署的时候环境不支持PATHINFO则改成兼容URL模式部署即可，程序和模板都不需要做任何改动。

注意：如果当前设置的是其他模式，但是URL里面出现了兼容模式的匹配参数，则会自动识别，也就是说兼容模式是优先判断的。

由于PATHINFO模式使用较多，所以后面的内容将主要以PATHINFO模式为例来说明。

[上一页](#)[下一页](#)

5.2 模块和操作

模块和操作

[上一页](#) [下一页](#)

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URL访问都可以认为是某个模块的某个操作，例如：

<http://www.domain.com/App/index.php/User/read/id/8>

<http://www.domain.com/index.php/Home/User/read/id/8>

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（Dispatcher）来实现，并且都分析成下面的规范：

[http://域名/项目名/分组名/模块名/操作名/其他参数](#)

Dispatcher会根据URL地址来获取当前需要执行的项目、分组（如果有定义的话）模块、操作以及其他参数，在某些情况下，项目名可能不会出现在URL地址中（通常情况下入口文件则代表了某个项目，而且入口文件可以被隐藏）。

每一个模块就是一个控制器类，通常位于项目的Lib\Action目录下。类名就是模块名加上Action后缀，例如UserAction类就表示了User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read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/App/index.php/User/read/id/8>

其实就是执行了UserAction类的read（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UserAction中如果没有定义help方法，但是存在对应的User/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/User/help/>

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

例外的情况就是如果定义了路由，则有可能URL的解析规则会被改变，这个我们会在URL路由中详细描述。

如果访问的URL是 <http://localhost/App/index.php>

在URL里面没有带任何模块和操作的参数，系统就会寻找默认模块DEFAULT_MODULE和默认操作DEFAULT_ACTION，系统默认的默认模块设置是Index模块，默认操作设置是index操作。也就是说：

<http://localhost/App/index.php>和

<http://localhost/App/index.php/Index>以及

<http://localhost/App/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。

如果我们访问一个不存在的操作或者模块，并且也没有渲染到默认定位的模板文件的话，在调试模式下面会抛出异常错误，在部署模式下则会发送404错误，但是可以通过空模块或者空操作方法引导这些页面到你希望的页面，请参考后面的空模块和空操作。3.1版本开始，增加ACTION_SUFFIX配置参数，用于设置操作方法的后缀。

例如，如果设置：`'ACTION_SUFFIX'=>'Act'` 那么访问某个模块的add操作对应读取模块类的操作方法则由原来的add方法变成addAct方法。

[上一页](#)[下一页](#)

5.3 定义控制器

定义控制器

[上一页](#)[下一页](#)

每个模块是一个Action文件，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，一般位于项目的Lib/Action目录下。

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：

每个模块是一个Action文件，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，一般位于项目的Lib/Action目录下。

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{} 控制器文件的名称是UserAction.class.php。
```

如果我们要执行下面的URL

<http://localhost/App/index.php/User/add>

则需要增加一个add操作方法就可以了，例如

控制器文件的名称是UserAction.class.php。

如果我们要执行下面的URL

<http://localhost/App/index.php/User/add>

```
<?php
//用户模块
class UserAction extends Action{
    //定义一个add操作方法
    public function add(){
        //add操作方法逻辑的实现
        // ...
        $this->display();//输出页面模板
    }
}
```

则需要增加一个add操作方法就可以了，例如

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\User\add.html

[上一页](#)[下一页](#)

5.4 空操作

空操作

[上一页](#)[下一页](#)

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，下面我们用空操作功能来实现一个城市切换的功能。

我们只需要给CityAction类定义一个_empty（空操作）方法：

```
<?php
class CityAction extends Action{
    public function _empty($name){
        //把所有城市的操作解析到city方法
        $this->city($name);
    }

    //注意 city方法 本身是 protected 方法
    protected function city($name){
        //和$name这个城市相关的处理
        echo '当前城市' . $name;
    }
}
```

接下来，我们就可以在浏览器里面输入

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

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

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

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

当前城市:beijing

当前城市:shanghai

当前城市:shenzhen

[上一页](#)[下一页](#)

5.5 空模块

空模块

[上一页](#) [下一页](#)

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

```
<?php
class EmptyAction extends Action{
    public function index(){
        //根据当前模块名来判断要执行那个城市的操作
        $cityName = MODULE_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;
    }
}
```

定义一个EmptyAction类

}

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/>

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

当前城市:beijing

当前城市:shanghai

当前城市:shenzhen

空模块和空操作还可以同时使用，用以完成更加复杂的操作。

[上一页](#) [下一页](#)

5.6 模块分组

模块分组

[上一页](#)[下一页](#)

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。

模块分组相关的配置参数包括：

配置参数	说明
APP_GROUP_LIST	项目分组列表（配置即表示开启分组）
DEFAULT_GROUP	默认分组（默认值为Home）
TMPL_FILE_DEPR	分组模板下面模块和操作的分隔符，默认值为“/”
VAR_GROUP	分组的URL参数名，默认为g（普通模式URL才需要）

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，那么只需要在项目配置中添加下面的配置：

```
'APP_GROUP_LIST' => 'Home,Admin', //项目分组设定
'DEFAULT_GROUP'  => 'Home', //默认分组
```

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> Home分组地址
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> Admin分组地址如果设置了隐藏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，分组和普通模块的项目目录区别如下：

项目目录	分组(以Home和Admin分组为例)	不分组
公共目录 (Common)	Home分组： Common/Home/function.php Admin分组： Common/Admin/function.php 公共文件：Common/common.php	Common/common.php

配置目录 （ Conf ）	Home分组：Conf/Home/config.php Admin分组： Conf/Admin/config.php 公共配置：Conf/config.php	Conf/config.php
Action目录	Home分组：Lib/Action/Home/ Admin分组：Lib/Action/Admin/ 公共Action：Lib/Action/	Lib/Action/
Model 目录	Lib/Model/	Lib/Model/
语言包目录（ Lang 以zh-cn为例 ）	Home分组：Lang/zh-cn/Home/lang.php Admin分组：Lang/zh-cn/Admin/lang.php 公共语言包：Lang/zh-cn/common.php	Lang/zh-cn/common.php
模板目录（ Tpl以theme主题为例 ）	Home分组：Tpl/Home/theme/ Admin分组：Tpl/Admin/theme/	Tpl/theme/
运行时目录（ Runtime ）	Home分组：Runtime/Home/ Admin分组：Runtime/Admin/	Runtime/

注意：分组目录的公共文件名称和语言包名称和公共的文件有一定的命名方式不同。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。

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/Home/Index/index.html

Tpl/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/Home/Index_index.html

Tpl/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。

注意：模块分组不支持配置不同的URL模式。从3.1版本开始，每个分组可以定义自己的空模块类EmptyAction。

[上一页](#)[下一页](#)

5.7 URL伪静态

URL伪静态

[上一页](#) [下一页](#)

ThinkPHP支持伪静态URL设置，可以通过设置URL_HTML_SUFFIX参数随意在URL的最后增加你想要的静态后缀，而不会影响当前操作的正常执行。例如，我们设置 'URL_HTML_SUFFIX'=>'shtml' 的话，我们可以把下面的URL `http://serverName/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：伪静态后缀设置时可以不包含后缀中的 “.”。所以，下面的配置其实是等效的：

'URL_HTML_SUFFIX'=>'.shtml' 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。

3.1版本开始，默认情况下，可以支持所有的静态后缀，并且会记录当前的伪静态后缀到常量EXT，但不会影响正常的页面访问。

```
http://serverName/User/3.html
http://serverName/User/3.shtml
http://serverName/User/3.xml
```

例如：`http://serverName/User/3.pdf` 都可以正常访问，如果要获取当前的伪静态后缀，通过常量EXT获取即可。

如果只是希望支持配置的伪静态后缀，可以直接设置成可以支持多个后缀，例如：

'URL_HTML_SUFFIX'=>'html|shmtl|xml' // 多个用 | 分割 那么，当访问

<http://serverName/User/3.pdf>的时候会报系统错误。

如果设置了多个伪静态后缀的话，使用U函数生成的URL地址中会默认使用第一个后缀，也支持指定后缀生成url地址。关于多伪静态后缀的支持

如果你希望网站能够支持多个伪静态后缀设置，例如，希望

```
http://serverName/Blog/read/id/1.shtml
http://serverName/Blog/read/id/1.html
http://serverName/Blog/read/id/1.xml
```

同时有效，可以用下面的方式进行配置：

'URL_HTML_SUFFIX'=>'(shtml|html|xml)' 配置多个伪静态后缀并不会导致自动判断后缀执行不同的方法，如果你有此类需求的话需要使用REST支持，可以参考第18张 REST支持部分。

[上一页](#) [下一页](#)

5.8 URL路由

URL路由

[上一页](#)[下一页](#)

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，并且配置URL_ROUTE_RULES参数后，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。

3.0版本的路由支持做了增强，包含规则路由和正则路由支持。

一、规则路由

规则路由是由2.1版本的简单路由进化而来，定义格式为：

格式1：'路由规则'=>'[分组/模块/操作]?额外参数1=值1&额外参数2=值2...'

格式2：'路由规则'=>array('[分组/模块/操作]','额外参数1=值1&额外参数2=值2...')

格式3：'路由规则'=>'外部地址'

格式4：'路由规则'=>array('外部地址','重定向代码')

注意事项：

- 路由规则中如果以 “:” 开头，表示动态变量，否则为静态地址
- 格式2的额外参数可以传入数组或者字符串
- 外部地址中如果要引用动态变量，采用 :1、:2 的方式
- 路由规则支持变量的数字约束定义，例如：'news/:id\d'=>'News/read'
- 规则路由可以支持 全动态和动静结合定义，例如:user/blog/:id'=>'Home/Blog/user'
- 路由规则非数字变量支持排除，例如 'news/:cate^add|edit|delete'=>'News/category'
- 路由规则中的静态地址部分不区分大小写

下面是规则路由的定义示例：

```
'URL_ROUTER_ON'    => true, //开启路由
'URL_ROUTE_RULES' => array( //定义路由规则
    'news/:year/:month/:day' => array('News/archive', 'status=1'),
    'news/:id'                => 'News/read',
    'news/read/:id'           => '/news/:1',
),
```

其中定义

`http://serverName/index.php/news/8`

了3条路由规则，如果我们访问下面的URL `http://serverName/index.php/news/10` 则会匹配到第二条规则路由，并解析到News模块的read操作，而且后面的数字会传入\$_GET['id']变量。

`http://serverName/index.php/news/2012/01/08`

如果我们访问下面的URL `http://serverName/index.php/news/2012/01/15` 则会匹配到第一条规则路由，并解析到News模块的archive操作，而且会传入year、month和day的GET变量。

第一条路由规则还可以改成 'news/:year/:month/:day/'=>'News/archive?status=1'，通常情况下，需要传入数组参数的时候才会需要使用格式数组来定义

第三条路由规则是一个路由重定向，一般是用于网站改版后的URL迁移，如果之前的URL访问规则是
<http://serverName/index.php/news/read/8> 那么会重定向到新的内部路由规则
<http://serverName/index.php/news/8> 这里之所以用了重定向路由是为了告诉搜索引擎这些地址已经发生改变了 而且以后是不需要保留。

有些情况下，可能会存在冲突，假如要支持通过标识来访问文章，

http://serverName/index.php/news/hello_world 那么解析规则就会混淆，但是我们可以更改路由规则如下：

```
'URL_ROUTER_ON'    => true, //开启路由
'URL_ROUTE_RULES' => array( //定义路由规则
    'news/:year/:month/:day' => array('News/archive', 'status=1'),
    'news/:id\d'           => 'News/read',
    'news/:name'           => 'News/read',
    'news/read/:id'        => '/news/:1',
),
```

news:id\d 规则表示当URL中id参数为数字时才会匹配

而 news/:name 规则定义 则会匹配所有的字符情况，这也是默认的情况，目前规则路由只区分数字和所有字符的情况，如果需要严格的类型约束，请采用正则路由定义规则。举个例子，我们现在用规则路由来实现之前用空操作实现的城市功能，我们定义了City控制器如下：

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

我们只需要定义下面的路由规则

'city/:name' => 'City/city' 就能实现之前用空操作实现的同样功能了。

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规则路由可以支持动态和静态混合甚至是全动态，例如：

```
'URL_ROUTER_ON'    => true, //开启路由
'URL_ROUTE_RULES' => array( //定义路由规则
    ':user/bolg/:id'    => 'Blog/read',
    ':user/:blog_name' => 'Blog/read',
),
```

第一条路由会匹配下列URL访问

<http://serverName/index.php/user1/blog/25/>

<http://serverName/index.php/username2/blog/245/>

并解析到Blog模块的read操作方法，传入user和id两个GET参数。

第二条路由会匹配到下面的URL访问

http://serverName/index.php/user1/hello_world

http://serverName/index.php/username2/test_nme

同样解析到Blog模块的read操作方法，只是传入的参数变成blog_name 一个GET参数。二、正则路由
正则路由可以实现更加复杂的路由定义，支持的定义格式如下：

格式1：'路由正则'=>'[分组/模块/操作]?参数1=值1&参数2=值2...'

格式2：'路由正则'=>array('[分组/模块/操作]', '参数1=值1&参数2=值2...')

格式3：'路由正则'=>'外部地址'

格式4：'路由正则'=>array('外部地址', '重定向代码')

注意事项：

- 正则路由规则必须以 "/" 开始和结束
- 格式2的参数可以传入数组或者字符串
- 参数值和外部地址中可以用动态变量 采用 :1、:2 的方式

下面是正则路由的定义示例：

```
'URL_ROUTER_ON'    => true, //开启路由
'URL_ROUTE_RULES' => array( //定义路由规则
    '/^blog\/(\d+)\$/'      => 'Blog/read?id=:1',
    '/^blog\/(\d+)\\/(\d+)\$/' => 'Blog/achive?year=:1&month=:2',
    '/^blog\/(\d+)\_(\d+)\$/' => 'blog.php?id=:1&page=:2',
),
```

[上一页](#) [下一页](#)

5.9 URL重写

URL重写

[上一页](#)[下一页](#)

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、把下面的内容保存为.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>
```

重启Apache之后，原来的

<http://serverName/index.php/Blog/read/id/1>

就可以通过访问

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

简化了URL地址。

[上一页](#)[下一页](#)

5.10 URL生成

URL生成

[上一页](#)[下一页](#)

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(' [分组/模块/操作]?参数' [, '参数', '伪静态后缀', '是否跳转', '显示域名']) 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('User/add') // 生成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
```

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

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

三种方式是等效的，都是 生成Blog模块的cate操作 并且cate_id为1 status为1的URL地址

但是不允许使用下面的定义方式来传参数 U('Blog/cate/cate_id/1/status/1') 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路由规则为： 'news/:id\d'=>'News/read' 那么可以使用 U('/news/1') 最终生成的URL地址是： <http://serverName/index.php/news/1>

注意：如果你是在模板文件中直接使用U方法的话，需要采用 {U('参数1', '参数2'...)} 的方式，具体参考模板引擎章节的8.3 使用函数内容。

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

U('Blog/read@blog.thinkphp.cn', 'id=1'); @后面传入需要指定的域名即可。

此外，U方法的第5个参数如果设置为true，表示自动识别当前的域名，并且会自动根据子域名部署设置

APP_SUB_DOMAIN_DEPLOY和APP_SUB_DOMAIN_RULES自动匹配生成当前地址的子域名。
如果开启了URL_CASE_INSENSITIVE，则会统一生成小写的URL地址。

[上一页](#)[下一页](#)

5.11 URL大小写

URL大小写

[上一页](#)[下一页](#)

我们知道，系统默认规范是根据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，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良好的解决方案，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

```
http://serverName/index.php/User/add
//将等效于
```

了。`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
```

 利用系统提供的U方法可以为你自动生成相关的URL地址。

如果设置 `'URL_CASE_INSENSITIVE' =>false` 的话，URL就又变成：

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

[上一页](#)[下一页](#)

5.12 前置和后置操作

前置和后置操作

[上一页](#)[下一页](#)

系统会检测当前操作是否具有前置和后置操作，如果存在就会按照顺序执行，前置和后置操作的方法名是

```
class CityAction extends Action{
    //前置操作方法
    public function _before_index(){
        echo 'before<br/>';
    }
    public function index(){
        echo 'index<br/>';
    }
    //后置操作方法
    public function _after_index(){
        echo 'after<br/>';
    }
}
```

在要执行的方法前面加 `_before_`和`after`，例如： }

如

果我们访问 `http://serverName/index.php/City/index` 结果会输出

before

index

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。如果当前的操作并没有定义操作方法，而是直接渲染模板文件，那么如果定义了前置 和后置方法的话，依然会生效。真正有模板输出的可能仅仅是当前的操作，前置和后置操作一般情况是没有任何输出的。

需要注意的是，在有些方法里面使用了`exit`或者错误输出之类的话 有可能不会再执行后置方法了。

例如，如果在当前操作里面调用了系统Action的`error`方法，那么将不会再执行后置操作，但是不影响`success`方法的后置方法执行。

[上一页](#)[下一页](#)

5.13 跨模块调用

跨模块调用

[上一页](#) [下一页](#)

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。

例如，我们在Index模块调用User模块的操作方法

```
class IndexAction extends Action{
    public function index(){
        //实例化UserAction
        $User = new UserAction();
        //其他用户操作
        //...
        $this->display(); //输出页面模板
    }
}
```

因为系统会自动加载Action控制器，因此 我们

不需要导入UserAction类就可以直接实例化。

并且为了方便跨模块调用，系统内置了A方法和R方法。

A方法表示实例化某个模块，例如，上面的方法可以改为：

```
class IndexAction extends Action{
    public function index(){
        //实例化UserAction
        $User = A('User');
        //其他用户操作
        //...
        $this->display(); //输出页面模板
    }
}
```

事实上，A方法还支持跨分组或者跨项目调用，

默认情况下是调用当前项目下面的模块。

跨项目调用的格式是：

A('[项目名:][分组名/]模块名')

A('User') //表示调用当前项目的User模块

A('Admin://User') //表示调用Admin项目的User模块

A('Admin/User') //表示调用Admin分组的User模块

例如：A('Admin://Tool/User') //表示调用Admin项目Tool分组的User模块 R方法表示调用一个模块的某个操作方法，调用格式是：

R('[项目名:][分组名/]模块名/操作名',array('参数1','参数2'...))

例如：

R('User/info') //表示调用当前项目的User模块的info操作方法

R('Admin/User/info') //表示调用Admin分组的User模块的info操作方法

R('Admin://Tool/User/info') //表示调用Admin项目Tool分组的User模块的info操作方法 R

方法还支持对调用的操作方法需要传入参数，例如User模块中我们定义了一个info方法：

```
class UserAction extends Action{
    protected function info($id){
        $User = M('User');
        $User->find($id);
        //...
    }
}
```

接下来，我们可以在其他模块中调用：

`R('User/info', array(15))` 表示调用当前项目的User模块的info操作方法，并且id参数传入15

[上一页](#)[下一页](#)

5.14 页面跳转

页面跳转

[上一页](#)[下一页](#)

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($result){
    //设置成功后跳转页面的地址，默认返回页面是$_SERVER['HTTP_REFERER']
    $this->success('新增成功', 'User/list');
} else {
    //错误页面的默认跳转页面是返回前一页，通常不需要设置
    $this->error('新增失败');
}
```

Success和

error方法都有对应的模板，并且是可以设置的，默认的设置是两个方法对应的模板都是：

```
//默认错误跳转对应的模板文件
'TMPL_ACTION_ERROR' => THINK_PATH . 'Tpl/dispatch_jump.tpl';
//默认成功跳转对应的模板文件
'TMPL_ACTION_SUCCESS' => THINK_PATH . 'Tpl/dispatch_jump.tpl'; 也可以使用项目内
//默认错误跳转对应的模板文件
'TMPL_ACTION_ERROR' => 'Public:error';
//默认成功跳转对应的模板文件
```

部的模板文件 'TMPL_ACTION_SUCCESS' => 'Public:success'; 模板文件可以使用模板标签，并且可以使用下面的模板变量：

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

success和error方法会自动判断当前请求是否属于Ajax请求，如果属于Ajax请求则会调用ajaxReturn方法返回信息，具体可以参考后面的AJAX返回部分。3.1版本开始，error和success方法支持传值，无论是跳转模板方式还是ajax方式 都可以使用assign方式传参。例如：

```
$this->assign('var1','value1');
$this->assign('var2','value2');
$this->error('错误的参数','要跳转的URL地址'); 当正常方式提交的时候，var1和var2变量会赋值到错误模板的模板变量。
```

当采用AJAX方式提交的时候，会自动调用ajaxReturn方法传值过去（包括跳转的URL地址url和状态值

status)

[上一页](#)[下一页](#)

5.15 重定向

重定向

[上一页](#)[下一页](#)

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：

```
//重定向到New模块的Category操作
$this->redirect('New/category', array('cate_id' => 2), 5, '页面跳转中...');
```

上面的用法是停留5秒后跳转到News模块的category操作，并且显示页面跳转中字样，重定向后会改变当前的URL地址。

如果你仅仅是想重定向到一个指定的URL地址，而不是到某个模块的操作方法，可以直接使用redirect方

```
//重定向到指定的URL地址
redirect('/New/category/cate_id/2', 5, '页面跳转中...')
```

Redirect方法的第一个参数是一个URL地址。

[上一页](#)[下一页](#)

5.16 获取系统变量

获取系统变量

[上一页](#)[下一页](#)

ThinkPHP没有改变原生的PHP系统变量获取方式，所以依然可以通过\$_GET、\$_POST、\$_SERVER、\$_REQUEST 等方式来获取系统变量，不过系统的Action类提供了对系统变量的增强获取方法，包括对GET、POST、PUT、REQUEST、SESSION、COOKIE、SERVER和GLOBALS参数，除了获取变量值外，还提供变量过滤和默认值支持，用法很简单，只需要在Action中调用下面方法：

`$this->方法名("变量名", ["过滤方法"], ["默认值"])` 方法名可以支持：

| 方法名 | 含义 |

|-----|-----|

| `_get` | 获取GET参数 |

| `_post` | 获取POST参数 |

| `_param` | 自动判断请求类型获取GET、POST或者PUT参数（3.1新增） |

| `_request` | 获取REQUEST 参数 |

| `_put` | 获取PUT 参数 |

| `_session` | 获取 `$_SESSION` 参数 |

| `_cookie` | 获取 `$_COOKIE` 参数 |

| `_server` | 获取 `$_SERVER` 参数 |

| `_globals` | 获取 `$GLOBALS`参数 |

变量名：（必须）是要获取的系统变量的名称

过滤方法：（可选）可以用任何的内置函数或者自定义函数名，如果没有指定的话，采用默认的 `htmlspecialchars` 函数进行安全过滤（由 `DEFAULT_FILTER` 参数配置），参数就是前面方法名获取到的值，也就是说如果调用：`$this->_get("name")`；最终调用的结果就是 `htmlspecialchars($_GET["name"])`，如果要改变过滤方法，可以使用：

`$this->_get("name", "strip_tags")`；默认值：（可选）是要获取的参数变量不存在的情况下设置的默认值，例如：`$this->_get("id", "strip_tags", 0)`；如果 `$_GET["id"]` 不存在的话，会返回0。

如果没有设置任何默认值的话，系统默认返回NULL。

其他方法的用法类似。也可以支持多函数过滤。

例如，可以设置：`'DEFAULT_FILTER'=>'htmlspecialchars,strip_tags'` 那么在控制器类如果调用 `$this->_get('id')`；的话，会依次对 `$_GET['id']` 变量进行 `htmlspecialchars` 和 `strip_tags` 方法过滤后返回结果。

下面调用方式也同样支持：`$this->_get('id', 'htmlspecialchars,strip_tags', 0)`；其他变量获取方法用法相同。

支持获取全部变量，例如：`$this->_get()`；表示获取`$_GET`变量值。

支持不过滤处理

```
$this->_get('id',false);
$this->_post('id',false);
//或者
$this->_get('id','');
```

如果不希望过滤某个参数，可以使用 `$this->_post('id','')`； 第二个参数使用false或者空字符串则表示不作任何过滤处理，即使我们有配置默认的过滤方法。

```
$this->_get('id');
```

如果我们忽略第二个参数调用的话 `$this->_post('id');` 则表示调用默认的过滤方法（由 `DEFAULT_FILTER` 参数进行配置）。3.1版本开始，Action类增加 `_param` 方法，可以自动根据当前请求类型（例如GET POST）获取参数。

例如：`$this->_param('id')`；当前为get方式提交的时候，就是获取`$_GET['id']`（进行默认过滤后）的值

当前为post方式提交的时候，就是获取`$_POST['id']`（进行默认过滤后）的值

还可以用 `_param` 方法获取URL中的参数

```
$this->_param(0); // 获取PATHINFO地址中的第一个参数
$this->_param(2); // 获取PATHINFO地址中的第3个参数
```

[上一页](#)[下一页](#)

5.17 判断请求类型

判断请求类型

[上一页](#) [下一页](#)

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

系统Action类内置了一些判断方法用于判断请求类型，包括：

| 方法 | 说明 |

|-----|-----|

| isGet | 判断是否是GET方式提交 |

| isPost | 判断是否是POST方式提交 |

| isPut | 判断是否是PUT方式提交 |

| isDelete | 判断是否是DELETE方式提交 |

| isHead | 判断是否是HEAD提交 |

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

使用举例如下：

另外还提供了一个判断当前是否

属于AJAX提交的方法

isAjax 是否属于AJAX提交

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

[上一页](#) [下一页](#)

5.18 获取URL参数

获取URL参数

[上一页](#) [下一页](#)

一般情况下URL中的参数就是通过GET方法获取，但是由于PATHINFO的特殊性，URL地址最终需要被解析才能转换成GET参数，ThinkPHP对URL是按照一定的规则进行解析的，除非你使用了URL路由规则，如果你对URL做了特别的定制，但是又不想使用URL路由，那么可以使用框架提供的URL参数获取方法直接获取，例如，我们访问一个如下的网址：

<http://serverName/News/archive/2012/01/15>

正常情况下，只有通过路由才能解析后面的2012/01/15，现在我们可以直接在News控制器的archive操

```
Class NewsAction extends Action {
    Public function archive(){
        $year    = $_GET["_URL"][2];
        $month   = $_GET["_URL"][3];
        $day     = $_GET["_URL"][4];
    }
}
```

作方法里面直接使用：

我们可以把URL地址

News/archive/2012/01/15 按照 “/” 分成多个参数，\$_GET["_URL"][0] 获取的就是News，\$_GET["_URL"][1]获取的就是archive，依次类推，可以通过数字索引获取所有的URL参数。3.0版开始支持URL地址中的PATH_INFO方式的URL的参数获取方式，需要配置

VAR_URL_PARAMS参数，默认值是：

```
'VAR_URL_PARAMS' => '_URL_', // PATHINFO URL参数变量 如果这个值不为空的话，就可以获取URL地址里面的PATH_INFO URL参数，例如
```

我们访问 <http://serverName.com/index.php/Blog/read/2012/03> 则可以在Blog控制器的

read操作方法里面采用

`$GET['_URL'][2]` 获取参数，表示获取PATH_INFO的URL参数

Blog/read/2012/03中的第3个参数（数组索引从0开始）

```
$year = $GET['_URL'][2]; // 2012
```

```
$month = $GET['_URL'][3]; // 03
```

3.1版本开始，建议使用_param方法获取URL参数，

_param方法方法是3.1新增的方法，可以自动根据当前请求类型获取参数。
_param方法的用法同_get和_post等方法，区别在于，_param方法能够自动根据当前请求类型自动获取相应的参数，例如：

如果当前是get请求方式，`$this->_param('id')`； 将会返回\$_GET['id'] 的处理数据

当采用POST请求方式的时候，同样的代码将会返回\$_POST['id']的处理数据

如果采用的是PUT请求，那么会自动返回PUT的处理数据，而无需开发人员进行判断。

并且需要注意的是，无论是什么方式的请求，系统都可以支持URL参数的获取，如果

```
$this->_param(1);
C('VAR_URL_PARAMS')设置不为空的话，就可以使用： $this->_param(2); 来获取URL地址中的
```

某个参数。 `$year = $this->_param(2);`
 `$month = $this->_param(3);` 的方式来获取。
 这样的好处是可以不需要使用路由功能就可以获取某个不规则的URL地址中的参数。

[上一页](#)[下一页](#)

5.19 AJAX返回

AJAX返回

[上一页](#) [下一页](#)

系统支持任何的AJAX类库，Action类提供了ajaxReturn方法用于AJAX调用后返回数据给客户端。并且支持JSON、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，默认配置采用JSON格式返回数据，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 |
```

调用示例：`$this->ajaxReturn(返回数据,提示信息,操作状态)`；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。

```
$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);
}
```

下面是一个简单的例子：

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。如果需要改变Ajax返回的数据格式，可以在控制器Action中增加ajaxAssign方法定义，定义格式如下：

```
public function ajaxAssign(&$result) {
    // 返回数据中增加url属性
    $result['url'] = $this->url;
}
```

3.1版本以后，ajaxReturn方法可以更加灵活的进

行ajax传值，并且废弃了ajaxAssign方法扩展。能够完全定义传值的数组和类型，例如：

```
$data['status'] = 1;
$data['info'] = 'info';
$data['size'] = 9;
$data['url'] = $url;
$this->ajaxReturn($data,'JSON'); data传值数组可以随意定义。
```

改进后的ajaxReturn方法也兼容之前的写法：`$this->ajaxReturn($data,'info',1)`；系统会自动

```
$data['info'] = 'info';
```

把info和1两个参数并入\$data数组中，等同于赋值

```
$data['status'] = 1;
```

[上一页](#)[下一页](#)

5.20 Action参数绑定

Action参数绑定

[上一页](#) [下一页](#)

Action参数绑定提供了URL变量和操作方法的参数绑定支持，这一功能可以使得你的操作方法定义和参数获取更加清晰，也便于跨模块调用了。这一新特性对以往的操作方法使用没有任何影响，你也可以用新的方式来改造以往的操作方法定义。

Action参数绑定的原理是把URL中的参数（不包括分组、模块和操作地址）和控制器的操作方法中的参数进行绑定。例如，我们给Blog模块定义了两个操作方法read和archive方法，由于read操作需要指定一个id参数，archive方法需要指定年份（year）和月份（month）两个参数。

```
`class BlogAction extends Action{
    public function read($id){
        echo 'id='.$id;
        $Blog = M('Blog');
        $Blog->find($id);
    }

    public function archive($year='2012',$month='01'){
        echo 'year='.$year.'&month='.$month;
        $Blog = M('Blog');
        $year = $year;
        $month = $month;
        $begin_time = strtotime($year . $month . "01");
        $end_time = strtotime("+1 month", $begin_time);
        $map['create_time'] = array(array('gt',$begin_time),array('lt',$end_time));
        $map['status'] = 1;
        $list = $Blog->where($map)->select();
    }
}
```

} URL的访问地址分别是：<http://serverName/index.php/Blog/read/id/5>

<http://serverName/index.php/Blog/archive/year/2012/month/03>两个URL地址中的id参数和year和month参数会自动和read操作方法以及archive操作方法的同名参数绑定。

id=5

输出的结果依次是：year=2012&month=03 Action参数绑定的参数必须和URL中传入的参数名称一致，但是参数顺序不需要一致。也就是说

<http://serverName/index.php/Blog/archive/month/03/year/2012> 和上面的访问结果是一致的，URL中的参数顺序和操作方法中的参数顺序都可以随意调整，关键是确保参数名称一致即可。

如果用户访问的URL地址是（至于为什么会这么访问暂且不提）：

`http://serverName/index.php/Blog/read/` 那么会抛出下面的异常提示：

参数错误:id

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

```
public function read($id=0){  
    echo 'id='.$id;  
    $Blog = M('Blog');  
    $Blog->find($id);  
}
```

认值，例如：

这样，当我们访问

`http://serverName/index.php/Blog/read/` 的时候 就会输出 `id=0` 当我们访问

`http://serverName/index.php/Blog/archive/` 的时候，输出：`year=2012&month=01`

[上一页](#)[下一页](#)

5.21 多层控制器支持

多层控制器支持

[上一页](#)[下一页](#)

3.1版本开始，控制器支持自定义分层。同时A方法增加第二个参数layer，用于设置控制器分层。

例如 `A('User', 'Event');` 表示实例化Lib/Event/UserEvent.class.php。

UserEvent如果继承Action类的话，可以使用Action类所有的功能。 `A('User', 'Api');` 表示实例化Lib/Api/UserApi.class.php

分层控制器仅用于内部调用，URL访问的控制器还是Action层，但是可以配置

DEFAULT_C_LAYER修改默认控制器层名称（该参数默认值为Action）。

[上一页](#)[下一页](#)

6. 模型

模型

[上一页](#)[下一页](#)

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扩展中。

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。

[上一页](#)[下一页](#)

6.1 模型定义

模型定义

[上一页](#)[下一页](#)

模型类一般位于项目的Lib/Model 目录下，当我们创建一个UserModel类的时候，其实已经遵循了系统的约定。模型类的命名规则是除去表前缀的数据表名称，采用驼峰法命名，并且首字母大写，然后加上模型类的后缀定义Model，例如：

模型名（类名）	约定对应数据表（假设数据库的前缀定义是 think_）
UserModel	think_user
UserTypeModel	think_user_type

如果你的规则和上面的系统约定不符合，那么需要设置Model类的数据表名称属性。
在ThinkPHP的模型里面，有几个关于数据表名称的属性定义：

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

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DBPREFIX 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（topdepts）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，例如：`protected $dbName = 'top';`

[上一页](#)[下一页](#)

6.2 模型实例化

模型实例化

[上一页](#) [下一页](#)

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

根据不同的模型定义，我们有几种实例化模型的方法，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：

```
//实例化User模型
$user = new Model('User');
//或者使用M()快捷方法实例化，和上面的方法是等效的
$user = M('User');
//执行其他的数据操作
$user->select();
```

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、实例化其他公共模型类

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。

`$user = new CommonModel('User');` 模型类的实例化方法有三个参数，第一个参数是模型名称，第二个参数用于设置数据表的前缀（留空则取当前项目配置的表前缀），第三个参数用于设置当前使用的数据库连接信息（留空则取当前项目配置的数据库连接信息），例如：

`$user = new CommonModel('User','think_','db_config');` 第三个连接信息参数可以使用DSN配置或者数组配置，甚至可以支持配置参数。关于这个参数的使用我们会在数据库连接部分详细描述。

用M方法实现的话，上面的方法可以写成：

`$user = M('CommonModel:User','think_','db_config');` M方法默认是实例化Model类，第二个参数用于指定表前缀，第三个参数就可以指定其他的数据库连接信息。

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。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、实例化用户自定义模型（xxxModel）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，

```
<?php
class UserModel extends Model{
    public function getTopUser(){
        //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。

```
<?php
//实例化自定义模型
$user = new UserModel();
//或者使用D快捷方法
$user = D('User');
//执行具体的数据操作
$user->select();
```

要实例化自定义模型类，可以使用下面的方式：

D方法可以自

动检测模型类，如果存在自定义的模型类，则实例化自定义模型类，如果不存在，则会实例化Model基类，同时对于已实例化过的模型，不会重复去实例化。

```
//实例化Admin项目的User模型
D('Admin://User')
//实例化Admin分组的User模型
```

D方法还可以支持跨项目和分组调用，需要使用：

```
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，

```
//实例化空模型
$model = new Model();
//或者使用M快捷方法是等效的
$model = M();
//进行原生的SQL查询
```

例如：`$model->query('SELECT * FROM think_user WHERE status = 1');` 空模型类也支持跨项目调用。我们在实例化的过程中，经常使用D方法和M方法，这两个方法的区别在于M方法实例化模型无需用户为每个数据表定义模型类，如果D方法没有找到定义的模型类，则会自动调用M方法。

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。

[上一页](#)[下一页](#)

6.3 字段定义

字段定义

[上一页](#) [下一页](#)

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

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

数据库名.模型名.php

例如：

thinkphp.User.php 表示User模型生成的字段缓存文件

thinkphp.Article.php 表示Article模型生成的字段缓存文件

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

从3.1版本开始，模型的字段缓存文件名全部转换成小写，避免重复生成。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 注意：调试模式下面由于考虑到数据结构可能会经常变动，所以默认是关闭字段缓存的。

如果需要显式获取当前数据表的字段信息，可以使用模型类的getDbFields方法来获取当前数据对象的全部字段信息，例如：`$fields = $User->getDbFields();`如果你在部署模式下面修改了数据表的字段信息，可能需要清空Data/_fields目录下面的缓存文件，让系统重新获取更新的数据表字段信息，否则会发生新增的字段无法写入数据库的问题。

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

```
<?php
class UserModel extends Model{
    protected $fields = array(
        'id', 'username', 'email', 'age', '_pk' => 'id', '_autoinc' => tr
    );
}
```

其中_pk 表示主键字段名称 _autoinc 表示主键是否自动增长类型，定义了fields属性之后，就不会自动获取数据表的字段信息了。如果有修改或者增加字段，必须手动修改fields属性的值。

[上一页](#) [下一页](#)

6.4 数据主键

数据主键

[上一页](#)[下一页](#)

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();
```

 注意：目前不支持联合主键的自动获取和操作。

[上一页](#)[下一页](#)

6.5 属性访问

属性访问

[上一页](#) [下一页](#)

ThinkPHP的模型对象实例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，可以支持对象和数组两种方式来访问数据属性，例如下面的方式采用数据对象的方式来访问User模型的属性：

```
//实例化User模型
$user = D('User');
//查询用户数据
$user->find(1);
//获取name属性的值
echo $user->name;
//设置name属性的值
$user->name = 'ThinkPHP';
```

除了find方法会产生数据对象属性外，data方法和create方法也会产生数据对象，例如：

```
$user = D('User');
$user->create();
```

echo \$user->name; 还有一种属性的操作方式是通过返回数组的方式：

```
//实例化User模型
$user = D('User');
//查询用户数据
$data = $user->find(1);
//获取name属性的值
echo $data['name'];
//设置name属性的值
$data['name'] = 'ThinkPHP';
```

两种方式的属性获取区别是一个是对象的属性，一个是数组的索引，开发人员可以根据自己的需要选择什么方式。

[上一页](#) [下一页](#)

6.6 跨库操作

跨库操作

[上一页](#)[下一页](#)

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

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

库info下面，那么我们只需要进行下面的设置即可。 } 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_'; 如果你没有定义模型类，而是使用的M方法操作的话，也可以支持跨库操作，例如：\$User = M('user.User', 'other_'); 表示实例化User模型，连接的是user数据库的other_user表。

[上一页](#)[下一页](#)

6.7 连接数据库

连接数据库

[上一页](#) [下一页](#)

ThinkPHP内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的Db类进行操作，而无需针对不同的数据库写不同的代码和底层实现，Db类会自动调用相应的数据库驱动来处理。目前的数据库包括Mysql、SqlServer、PgSQL、Sqlite、Oracle、Ibase、Mongo，也包括对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。

常用的配置方式是在项目配置文件中添加下面的参数：

```
<?php
//项目配置文件
return array(
    //数据库配置信息
    'DB_TYPE'    => 'mysql', // 数据库类型
    'DB_HOST'    => 'localhost', // 服务器地址
    'DB_NAME'    => 'thinkphp', // 数据库名
    'DB_USER'    => 'root', // 用户名
    'DB_PWD'     => '', // 密码
    'DB_PORT'    => 3306, // 端口
    'DB_PREFIX'  => 'think_', // 数据库表前缀
    //其他项目配置参数
    // ...
);
```

或者采用如下配置

'DB_DSN' => 'mysql://username:password@localhost:3306/DbName' 使用DB_DSN方式定义可以简化配置参数，DSN参数格式为：

****数据库类型://用户名:密码@数据库地址:数据库端口/数据库名**** 如果两种配置参数同时存在的话，DB_DSN配置参数优先。

注意：如果要设置分布式数据库，暂时不支持DB_DSN方式配置。如果采用PDO驱动的话，则必须首先配置DB_TYPE 为pdo，然后还需要单独配置其他参数，例如：

```
//PDO连接方式
'DB_TYPE'    => 'pdo', // 数据库类型
'DB_USER'    => 'root', // 用户名
'DB_PWD'     => '', // 密码
'DB_PREFIX'  => 'think_', // 数据库表前缀
'DB_DSN'     => 'mysql:host=localhost;dbname=thinkphp;charset=UTF-8' 注意：PDO
```

方式的DB_DSN配置格式有所区别，根据不同的数据库类型设置有所不同。

配置文件定义的数据库连接信息一般是系统默认采用的，因为一般一个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。

可以对每个项目和不同的分组定义不同的数据库连接信息，如果开启了调试模式的话，还可以在不同的应用状态的配置文件里面定义独立的数据库配置信息。第二种 在模型类里面定义connection属性

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

```
//在模型里单独设置数据库连接信息
protected $connection = array(
    'db_type'    => 'mysql',
    'db_user'    => 'root',
    'db_pwd'     => '1234',
    'db_host'    => 'localhost',
    'db_port'    => '3306',
    'db_name'    => 'thinkphp'
```

据库，例如：);

也可以采用DSN方式定义，例如：

//或者使用DSN定义

protected \$connection = 'mysql://root:1234@localhost:3306/thinkphp'; 如果我们已经在配置文件中配置了额外的数据库连接信息，例如：

//数据库配置1

```
'DB_CONFIG1' = array(
    'db_type'    => 'mysql',
    'db_user'    => 'root',
    'db_pwd'     => '1234',
    'db_host'    => 'localhost',
    'db_port'    => '3306',
    'db_name'    => 'thinkphp'
```

),

//数据库配置2

```
'DB_CONFIG2' => 'mysql://root:1234@localhost:3306/thinkphp'; 那么，我们可以把模
//调用配置文件中的数据库配置1
protected $connection = 'DB_CONFIG1';
//调用配置文件中的数据库配置2
```

型类的属性定义改为：protected \$connection = 'DB_CONFIG2'; 如果采用的是M方法实例化模型的话，也可以支持传入不同的数据库连接信息，例如：

\$User = M('User','other_', 'mysql://root:1234@localhost/demo'); 表示实例化User模型，连接的是demo数据库的other_user表，采用的连接信息是第三个参数配置的。如果我们在项目配置文件中已经配置了DBCONFIG2的话，也可以采用：`\$User = M('User','other','DB_CONFIG2');`

如果你的个别数据表没有定义任何前缀的话，可以在前缀参数中传入NULL，例如：

\$User = M('User',Null,'DB_CONFIG2');`表示实例化User模型，连接的是demo数据库的用户表。

需要注意的是，ThinkPHP的数据库连接是惰性的，所以并不是在实例化的时候就连接数据库，而是在有实际的数据操作的时候才会去连接数据库（额外的情况是，在系统第一次实例化模型的时候，会自动连接数据库获取相关模型类对应的数据表的字段信息）。

[上一页](#)[下一页](#)

6.8 切换数据库

切换数据库

[上一页](#) [下一页](#)

如果你需要切换到另外一个数据库（包括在相同和不同的数据库类型之间切换）或者需要连接多个数据库进行操作不同的数据，就需要使用ThinkPHP提供的数据库切换方法，用法很简单，只需要调用Model类的db方法，用法：`Model->db("数据库编号", "数据库配置")`；数据库编号用数字格式，对于已经调用过的数据库连接，是不需要再传入数据库连接信息的，系统会自动记录。对于默认的数据库连接，内部的数据库编号是0，因此为了避免冲突，请不要再次定义数据库编号为0的数据库配置。

数据库配置的定义方式和模型定义connection属性一样，支持数组、字符串以及调用配置参数三种格式。Db方法调用后返回当前的模型实例，直接可以继续进行模型的其他操作，所以该方法可以在查询的过程中动态切换，例如：

`$this->db(1, "mysql://root:123456@localhost:3306/test")->query("查询SQL")`；该方法添加了一个编号为1的数据库连接，并自动切换到当前的数据库连接。

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

`$this->db(1)->query("查询SQL")`；如果需要切换到默认的数据库连接，只需要调用：

`$this->db(0)`；如果我们已经在项目配置中定义了其他的数据库连接信息，例如：

//数据库配置1

```
'DB_CONFIG1' = array(
    'db_type'    => 'mysql',
    'db_user'    => 'root',
    'db_pwd'     => '1234',
    'db_host'    => 'localhost',
    'db_port'    => '3306',
    'db_name'    => 'thinkphp'
),
```

//数据库配置2

```
'DB_CONFIG2' => 'mysql://root:1234@localhost:3306/thinkphp'; 我们就可以直接在db
    $this->db(1, "DB_CONFIG1")->query("查询SQL");
```

方法中调用配置进行连接了：`$this->db(2, "DB_CONFIG2")->query("查询SQL")`；如果切换数据库之后，数据表和当前不一致的话，可以使用table方法指定要操作的数据表：

`$this->db(1)->table("top_user")->find()`；我们也可以直接用M方法切换数据库，例如：

`M("User", "think_", "mysql://root:123456@localhost:3306/test")->query("查询SQL")`；

或者 `M("User", "think_", "DB_CONFIG1")->query("查询SQL")`；

[上一页](#) [下一页](#)

6.9 分布式数据库

分布式数据库

[上一页](#) [下一页](#)

ThinkPHP内置了分布式数据库的支持，包括主从式数据库的读写分离，但是分布式数据库必须是相同的数据库类型。配置DB_DEPLOY_TYPE 为1 可以采用分布式数据库支持。如果采用分布式数据库，定义数

```
//在项目配置文件里面定义
return array(
    //分布式数据库配置定义
    'DB_TYPE'    => 'mysql', //分布式数据库类型必须相同
    'DB_HOST'    => '192.168.0.1,192.168.0.2',
    'DB_NAME'    => 'thinkphp', //如果相同可以不用定义多个
    'DB_USER'    => 'user1,user2',
    'DB_PWD'     => 'pwd1,pwd2',
    'DB_PORT'    => '3306',
    'DB_PREFIX'  => 'think_',
    //其他配置参数
    // ...
);
```

数据库配置信息的方式如下：

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_USER'=>'user1',
'DB_PORT'=>'3306' 等效。 'DB_PWD'=>'pwd1', 和
'DB_USER'=>'user1,user1',
'DB_PWD'=>'pwd1,pwd1', 等效。还可以设置分布式数据库的读写是否分离，默认的情况下

读写不分离，也就是每台服务器都可以进行读写操作，对于主从式数据库而言，需要设置读写分离，通过下面的设置就可以： 'DB_RW_SEPARATE'=>true, 在读写分离的情况下，默认第一个数据库配置是主服务器的配置信息，负责写入数据，如果设置了DB_MASTER_NUM参数，则可以支持多个主服务器写入。其它的都是从数据库的配置信息，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随机进行在从服务器中选择。

CURD操作系统会自动判断当前执行的方法的读操作还是写操作，如果你用的是原生SQL，那么需要注意系统的默认规则：

写操作必须用模型的execute方法，读操作必须用模型的query方法，否则会发生主从读写错乱的情况。

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

[上一页](#) [下一页](#)

6.10 创建数据

创建数据

[上一页](#) [下一页](#)

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。

```
// 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
```

很简单的例子： `$user->add();`

Create方法支持从其它方式创建数据对

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'ThinkPHP@gmail.com';
```

象，例如，从其它的数据对象，或者数组等 `$user->create($data);`

甚至

```
// 从User数据对象创建新的Member数据对象
$user = M("User");
$user->find(1);
$member = M("Member");
```

还可以支持从对象创建新的数据对象 `$member->create($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，完成了一系列的工作，我们来看下create方法的工作流程就能明白：

| 步骤 | 说明 | 返回 |

|-----|-----|-----|

- | 1 | 获取数据源（默认是POST数组） | |
- | 2 | 验证数据源合法性（非数组或者对象会过滤） | 失败则返回false |
- | 3 | 检查字段映射 | |
- | 4 | 判断提交状态（新增或者编辑 根据主键自动判断） | |
- | 5 | 数据自动验证 | 失败则返回false |
- | 6 | 表单令牌验证 | 失败则返回false |
- | 7 | 表单数据赋值（过滤非法字段和字符串处理） | |
- | 8 | 数据自动完成 | |
- | 9 | 生成数据对象（保存在内存） | |

因此，我们熟悉的令牌验证、自动验证和自动完成（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方法才会真正写入数据库。

因此在没有调用add或者save方法之前，我们都可以改变create方法创建的数据对象，例如：

```
$User = M('User');
$user->create(); //创建User数据对象
$user->status = 1; // 设置默认的用户状态
$user->create_time = time(); // 设置用户的创建时间
$user->add(); // 把用户对象写入数据库
```

如果只是想简单创建一个数据对象，

并不需要完成一些额外的功能的话，可以使用data方法简单的创建数据对象。

```
// 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';
```

使用如下：`$user->data($data)->add();`

Data方法也支持传入数组和对象，使用

data方法创建的数据对象不会进行自动验证和过滤操作，请自行处理。但在进行add或者save操作的时候，数据表中不存在的字段以及非法的数据类型（例如对象、数组等非标量数据）是会自动过滤的，不用担心非数据表字段的写入导致SQL错误的问题。安全提示：

create方法如果没有传值，默认取\$_POST数据，如果用户提交的变量内容，含有可执行的html代码，请进行手工过滤。`$_POST['title'] = "<script>alert(1);</script>";`非法html代码可以使用htmlspecialchars进行编码，以防止用户提交的html代码在展示时被执行，以下是两种安全处理方法。

```
$_POST['title'] = htmlspecialchars($_POST['title']);
M('User')->create();
$data['title'] = $this->_post('title', 'htmlspecialchars');
M('User')->create($data);
```

如果要对全局变量进

行修改，请参考本手册14.4 输入过滤(VAR_FILTERS参数)和 [TP入门系列之安全设置](#)

[上一页](#)[下一页](#)

6.11 字段映射

字段映射

[上一页](#)[下一页](#)

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

```
Class UserModel extends Model{
    protected $_map = array(
        'name' =>'username', // 把表单中name映射到数据表的username字段
        'mail'  =>'email',   // 把表单中的mail映射到数据表的email字段
    );
}
```

下： } 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实际数据表字段。字段映射还可以支持对主键的映射。

如果我们需要把数据库中的数据显示在表单中，并且也支持字段映射的话，需要对查询的数据进行一下处

```
// 实例化User模型
$user = M('User');
```

理，处理方式是调用Model类的parseFieldsMap方法，例如：\$data = \$user->find(3); 这个时候取出的data数据包含的是实际的username和email字段，为了方便便表单输出，我们需要处理成字段映射显示在表单中，就需要使用下面的代码处理：\$data = \$user->parseFieldsMap(\$data); 这样一来，data数据中就包含了name和mail字段数据了，而不再有username和email字段数据了。

[上一页](#)[下一页](#)

6.12 连贯操作

连贯操作

[上一页](#) [下一页](#)

ThinkPHP模型基础类提供的连贯操作方法，可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的CURD操作。使用也比较简单，假如我们现在要查询一个User表的满足状态为1的前10条记录，并希望按照用户的创建时间排序，代码如下：

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

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

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

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

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

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

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

```
$User->where('status=1 and id=1')->delete();
```

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

系统支持的连贯操作方法有：

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

本文档使用 [看云](#) 构建

| relation | 用于关联查询（需要关联模型支持） | 字符串 |
所有的连贯操作都返回当前的模型实例对象（this），其中带*标识的表示支持多次调用。

WHERE

where 用于查询或者更新条件的定义	
用法	where(\$where)
参数	where（必须）：查询或者操作条件，支持字符串、数组和对象
返回值	当前模型实例
备注	如果不调用where方法，默认不会执行更新和删除操作

Where方法是使用最多的连贯操作方法，更详细的用法请参考后面的6.13 CURD操作和6.18查询语言部分。

TABLE

table 定义要操作的数据表名称，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，可以使用别名和跨库操作	
用法	table(\$table,\$parse=null)
参数	table（必须）：数据表名称，支持操作多个表，支持字符串、数组和对象 parse（可选）预处理参数，详见14.3防止SQL注入 查询条件预处理
返回值	当前模型实例
备注	如果不调用table方法，会自动获取模型对应或者定义的数据表

用法示例：`$Model->Table('think_user user')->where('status>1')->select();`也可以在table方法中跨库操作，例如：
`$Model->Table('db_name.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('sta`
使用数组方式定义的优势是可以避免因为表名和关键字冲突而出错的情况。
一般情况下，无需调用table方法，默认会自动获取当前模型对应或者定义的数据表。

DATA

data 可以用于新增或者保存数据之前的数据对象赋值	
用法	data(\$data)
参数	data（必须）：数据，支持数组和对象
返回值	当前模型实例
备注	如果不调用data方法，则会取当前的数据对象或者传入add和save的数据

`$Model->data($data)->add();`
使用示例：`$Model->data($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
本文档使用 [看云](#) 构建

象赋值的方式。

模型的data方法除了创建数据对象之外，还可以读取当前的数据对象，

```
$this->find(3);
```

例如：`$data = $this->data();`

FIELD

field 用于定义要查询的字段	
用法	field(\$field,\$except=false)
参数	field（必须）：字段名，支持字符串和数组，支持指定字段别名；如果为true则表示显式或者数据表的所有字段。 except（可选）：是否排除，默认为false，如果为true表示定义的字段为数据表中排除field参数定义之外的所有字段。
返回值	当前模型实例
备注	如果不调用field方法，则默认返回所有字段，和field（'*'）等效

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

使用示例：`$Model->field(array('id','nickname'=>'name'))->select();` 如果不调用field方法或者field方法传入参数为空的话，和使用field（'*'）是等效的。

如果需要显式的传入所有的字段，可以使用下面的方法：`$Model->field(true)->select();` 但是我们更建议只获取需要显式的字段名，或者采用字段排除方式来定义，例如：

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

 表示获取除了status之外的所有字段。

ORDER

order 用于对操作结果排序	
用法	order(\$order)
参数	order（必须）：排序的字段名，支持字符串和数组，支持多个字段排序
返回值	当前模型实例
备注	如果不调用order方法，按照数据库的默认规则

使用示例：`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

limit 用于定义要查询的结果限制（支持所有的数据库类型）	
用法	limit(\$limit)

参数	limit (必须) : 限制数量, 支持字符串
返回值	当前模型实例
备注	如果不调用limit方法, 则表示没有限制

备注 如果不调用limit方法, 则表示没有限制

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。

使用示例: `limit('1,10')` 如果使用`limit('10')` 等效于 `limit('0,10')`

3.1版本以后, limit方法增加第二个参数支持, 例如: `$this->limit(10,100)->select();` 和之前的用法 `$this->limit('10,100')->select();` 等效。

PAGE

page	用于定义要查询的数据分页
用法	page(\$page)
参数	page (必须) : 分页, 支持字符串
返回值	当前模型实例
备注	无

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, 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, 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。

3.1版本以后, page方法增加第二个参数支持, 例如: `$this->page(5,25)->select();` 和之前的用法 `$this->page('5,25')->select();` 等效。

GROUP

group	用于数据库的group查询支持
用法	group(\$group)
参数	group (必须) : group的字段名, 支持字符串
返回值	当前模型实例
备注	无

使用示例: `group('user_id')` Group方法的参数只支持字符串

HAVING

having 用于数据库的having查询支持	
用法	having(\$having)
参数	having (必须) : having , 支持字符串
返回值	当前模型实例
备注	无

使用示例：`having('user_id>0')` having方法的参数只支持字符串

JOIN

join 用于数据库的join查询支持	
用法	join(\$join)
参数	join (必须) : join操作，支持字符串和数组
返回值	当前模型实例
备注	join方法支持多次调用

使用示例：
`$Model->join(' work ON artist.id = work.artist_id')->join('card ON artist.car`
默认采用LEFT JOIN 方式，如果需要其他的JOIN方式，可以改成
`$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 = ca`

UNION

union 用于数据库的union查询支持	
用法	union(\$union,\$all=false)
参数	union (必须) : union操作，支持字符串、数组和对象 all (可选) : 是否采用UNION ALL 操作，默认为false
返回值	当前模型实例
备注	Union方法支持多次调用

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

使用示例：

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

数组用法：

或者


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

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

支持UNION ALL 操作，例如：

或者

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

每个union方法相当于一个独立的SELECT语句。

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

DISTINCT

distinct 查询数据的时候进行唯一过滤	
用法	distinct(\$distinct)
参数	distinct（必须）：是否采用distinct，支持布尔值
返回值	当前模型实例
备注	无

使用示例：`$Model->Distinct(true)->field('name')->select();`

RELATION

relation 用于关联查询支持	
用法	relation(\$relation)
参数	relation（必须）：关联操作
返回值	当前模型实例
备注	使用关联模型才支持

关联查询方法的详细用法请参考后面的6.23关联模型部分。

LOCK

lock 用于查询或者写入锁定	
用法	lock(\$lock)
参数	lock（必须）：是否需要锁定，支持布尔值
返回值	当前模型实例

备注	join方法支持多次调用
----	--------------

Lock方法是用于数据库的锁机制，如果在查询或者执行操作的时候使用：`lock(true)` 就会自动在生成的SQL语句最后加上 `FOR UPDATE`或者`FOR UPDATE NOWAIT`（Oracle数据库）。

CACHE

cache 用于查询缓存操作	
用法	<code>cache(\$key=true,\$expire=",\$type=")</code>
参数	<code>key</code> （可选）：是否启用查询缓存，支持布尔值和字符串，如果是字符串表示查询缓存的缓存名 <code>expire</code> （可选）：查询缓存的有效期，如果留空取系统默认的缓存有效期 <code>type</code> （可选）：查询缓存的缓存类型，如果留空取系统默认的缓存类型
返回值	当前模型实例
备注	如果不调用 <code>field</code> 方法，则默认返回所有字段，和 <code>field（'*'）</code> 等效

查询缓存的详细用法会在后面的12.6查询缓存部分详细描述。

[上一页](#)[下一页](#)

6.13 CURD操作

CURD操作

[上一页](#)[下一页](#)

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）

创建（Create）

在ThinkPHP中使用add方法新增数据到数据库（而并不是create方法）。

add 写入（新增）数据到数据库	
用法	add(\$data="", \$options=array(), \$replace=false)
参数	data（可选）：要新增的数据，支持数组和对象，如果留空取当前数据对象 options（可选）：操作表达式，通常由连贯操作完成，默认为空数组 replace（可选）：是否允许写入时更新，默认为false（个别数据库支持）
回调接口	写入前 _before_insert(&\$data, \$options) 写入成功 _after_insert(\$data, \$options)
返回值	如果数据非法或者查询错误则返回false 如果是自增主键 则返回主键值，否则返回1
相关方法	通常和data、create方法配合使用

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

使用示例如下：\$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。

```
$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();
```

使用create方法的例子：\$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。

从2.1版开始恢复了批量插入数据的addAll方法（仅针对Mysql数据库），如：

本文档使用 [看云](#) 构建

`$User->addAll($data)` 同时在数据插入时允许更新操作:
`add($data='', $options=array(), $replace=false)` 其中add方法增加\$replace参数(是否添加数据时允许覆盖), true表示覆盖, 默认为false

读取 (Read)

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select方法 (新版已经废除原来的findall方法) :

select 查询数据集	
用法	<code>select(\$options=array())</code>
参数	options (可选) : 为数组的时候表示操作表达式, 通常由连贯操作完成; 如果是数字或者字符串, 表示主键值。默认为空数组。
回调接口	查询成功 <code>_after_select(&\$resultSet,\$options)</code>
返回值	查询错误返回false 查询结果为空返回null 查询成功返回查询的结果集 (二维索引数组)
相关方法	通常配合连贯操作where、field、order、limit、join等一起使用

使用示例 :

```
$User = M("User"); // 实例化User对象
// 查找status值为1的用户数据 以创建时间排序 返回10条数据
$list = $User->where('status=1')->order('create_time')->limit(10)->select();
```

Select方法配合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 :

find 查询数据	
用法	<code>find(\$options=array())</code>
参数	options (可选) : 为数组的时候表示操作表达式, 通常由连贯操作完成; 为数字或者字符串的时候表示主键值。默认为空数组。
回调接口	查询后 <code>_after_find(&\$result,\$options)</code>
返回值	如果查询错误返回false 如果查询结果为空返回null 如果查询成功返回查询的结果 (索引数组)

相关方法	通常配合连贯操作where、field、order、join等一起使用
------	-------------------------------------

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
```

下面是一些查询的例子：`$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

getField 查询某个字段的值	
用法	getField(\$field,\$sepa=null)
参数	field（必须）：要获取的字段字符串（多个用逗号分隔） sepa（可选）：字段数据间隔符号，如果是 NULL返回数组为数组。默认为null。
回调接口	查询后 _after_find(&\$result,\$options)
返回值	如果查询结果为空返回null 如果field是一个字段则返回该字段的值 如果field是多个字段，返回数组。数组的索引是第一个字段的值，sepa为null则返回二维数组。
相关方法	通常配合连贯操作where、limit、order等一起使用

```
$User = M("User"); // 实例化User对象
// 获取ID为3的用户的昵称
```

示例如下：`$nickname = $User->where('id=3')->getField('nickname');`当只有一个字段的时候，默认返回一个值。

如果需要返回数组，可以用：`$this->getField('id',true);` // 获取id数组 如果传入多个字段

```
$User = M("User"); // 实例化User对象
// 获取所有用户的ID和昵称列表
```

的话，默认返回一个关联数组：`$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。

如果传入多个字段的名称，例如：`$list = $User->getField('id,nickname,email');`返回的是一个二维数组，类似select方法的返回结果，区别的是这个二维数组的键名是用户的id（准确的说是getField方法的第一个字段名）。

如果我们传入一个字符串分隔符：`$list = $User->getField('id,nickname,email',':');`那么返回的结果就是一个数组，键名是用户id，键值是 nickname:email的输出字符串。

getField方法的sepa参数还可以支持限制数量，例如：

```
$this->getField('id,name',5); // 限制返回5条记录
$this->getField('id',3); // 获取id数组 限制3条记录
```

 可以配合使用order方法使用。

更新 (Update)

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。

save 更新数据到数据库	
用法	save(\$data="", \$options=array())
参数	data：要保存的数据，如果为空，则取当前的数据对象。 options：为数组的时候表示操作表达式，通常由连贯操作完成；为数字或者字符串的时候表示主键值。默认为空数组。
回调接口	更新前_before_update(&\$data,\$options) 更新成功后 _after_update(\$data,\$options)
返回值	如果查询错误或者数据非法返回false 如果更新成功返回影响的记录数
相关方法	通常配合连贯操作where、field、order等一起使用

```
$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
    $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();
```

的例子：\$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。

setField 更新某个字段的值	
用法	setField(\$field,\$value="")
参数	options (可选)：为数组的时候表示操作表达式，通常由连贯操作完成；为数字或者字符串的时候表示主键值。默认为空数组。
返回值	如果查询错误返回false 如果更新成功返回影响的记录数
相关方法	必须配合连贯操作where一起使用

```
$User = M("User"); // 实例化User对象
// 更改用户的name值
```

使用示例：\$User-> where('id=5')->setField('name','ThinkPHP'); setField方法支持同时更新多个字段，只需要传入数组即可，例如：

```
$User = M("User"); // 实例化User对象
// 更改用户的name和email的值
$data = array('name'=>'ThinkPHP','email'=>'ThinkPHP@gmail.com');
$User-> where('id=5')->setField($data);
```

而对于统计字

段（通常指的是数字类型）的更新，系统还提供了setInc和setDec方法。

setInc /setDec 字段增长/字段减少	
用法	setInc(\$field,\$step=1)字段值增长 setDec(\$field,\$step=1)字段值减少
参数	field：要更新的字段名。 step：增长或者减少的数值，默认为1。
回调接口	如果查询错误返回false 如果更新成功返回影响的记录数
返回值	如果查询错误返回false

	如果更新成功返回影响的记录数
相关方法	必须配合连贯操作where一起使用

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

删除 (Delete)

在ThinkPHP中使用delete方法删除数据库中的记录。

用法	delete(\$options=array())
参数	options：为数组的时候表示操作表达式，通常由连贯操作完成，如果没有传入任何删除条件，则取当前数据对象的主键作为条件；为数字或者字符串的时候表示主键值。默认为空数组。
回调接口	删除成功后 _after_delete(\$data,\$options)
返回值	如果查询错误返回false 如果删除成功返回影响的记录数
相关方法	通常配合连贯操作where、field、order等一起使用

```
$User = M("User"); // 实例化User对象
$User->where('id=5')->delete(); // 删除id为5的用户数据
```

示例如下：\$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();
```

[上一页](#)[下一页](#)

6.14 ActiveRecord

ActiveRecord

[上一页](#) [下一页](#)

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。

```
$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
```

一、创建数据 \$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();
```

二、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

```
$User = M("User"); // 实例化User对象
// 查找id为8的用户数据
```

照之前的方式，我们可能会使用下面的方法：\$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
```

用：\$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

```
echo $User->name;
```

取查询的结果数据：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');
```

三、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

```
$User->id = 1;
$User->name = 'TOPThink'; // 修改数据对象
```

一定要先查询。因为下面的方式其实是等效的：\$User->save(); // 保存当前数据对象

四、删除记录

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

可以删除当前查询的数据对象 `$User->delete();` // 删除当前的数据对象 或者直接根据主键进行删

```
$User->delete(8); // 删除主键为8的数据
```

除 `$User->delete('5,6');` // 删除主键为5、6的多个数据

[上一页](#)[下一页](#)

6.15 自动验证

自动验证

[上一页](#)[下一页](#)

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二维数组。

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])` 说明

验证 字段	必须	需要验证的表单字段名称，这个字段不一定是数据库字段，也可以是表单的一些辅助字段，例如确认密码和验证码等等。有个别验证规则和字段无关的情况下，验证字段是可以随意设置的，例如expire有效期规则是和表单字段无关的。
验证 规则	必须	要进行验证的规则，需要结合附加规则，如果在使用正则验证的附加规则情况下，系统还内置了一些常用正则验证的规则，可以直接作为验证规则使用，包括： <code>require</code> 字段必须、 <code>email</code> 邮箱、 <code>url</code> URL地址、 <code>currency</code> 货币、 <code>number</code> 数字。
提示 信息	必须	用于验证失败后的提示信息定义
验证 条件	可选	包含下面几种情况： <code>Model::EXISTS_VALIDATE</code> 或者0 存在字段就验证（默认） <code>Model::MUST_VALIDATE</code> 或者1 必须验证 <code>Model::VALUE_VALIDATE</code> 或者2 值不为空的时候验证
		配合验证规则使用，包括下面一些规则： <code>regex</code> 正则验证，定义的验证规则是一个正则表达式（默认） <code>function</code> 函数验证，定义的验证规则是一个函数名 <code>callback</code> 方法验证，定义的验证规则是当前模型类的一个方法 <code>confirm</code> 验证表单中的两个字段是否相同，定义的验证规则是一个字段名 <code>equal</code> 验证是否等于某个值，该值由前面的验证规则定义 <code>in</code> 验证是否在某个范围内，定义的验证规则必须是一个数组

附加规则	可选	 length验证长度，定义的验证规则可以是一个数字（表示固定长度）或者数字范围（例如3,12 表示长度从3到12的范围） between验证范围，定义的验证规则表示范围，可以使用字符串或者数组，例如1,31或者array(1,31) expire验证是否在有效期，定义的验证规则表示时间范围，可以到时间，例如可以使用 2012-1-15,2013-1-15 表示当前提交有效期在2012-1-15到2013-1-15之间，也可以使用时间戳定义 ip_allow 验证IP是否允许，定义的验证规则表示允许的IP地址列表，用逗号分隔，例如 201.12.2.5,201.12.2.6 ip_deny 验证IP是否禁止，定义的验证规则表示禁止的ip地址列表，用逗号分隔，例如 201.12.2.5,201.12.2.6 unique 验证是否唯一，系统会根据字段目前的值查询数据库来判断是否存在相同的值。
验证时	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
    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
    array('repassword','password','确认密码不正确',0,'confirm'), // 验证确认密码是
    array('password','checkPwd','密码格式不正确',0,'function'), // 自定义函数验证
);
```

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
}

```

多字段验

证

自动验证功能中的function和callback规则可以支持多字段。

例子：`protected \$_validate = array(
 array('user_id,good_id','checkIfOrderToday','今天已经购买过，请明天再来',1,'callback',1),
);

```

protected function checkIfOrderToday($data){
    $map = $data;
    $map['ctime'] = array(array('gt',[开始时间]), array('lt', [结束时间]));
    if($this->where($map)->find())
        return false;
    else
        return true;
}

```

****批量验证****

新版支持数据的批量验证功能，只需要在模型类里面设置patchValidate属性为true（默认为false）
 array("字段名1"=>"错误提示1","字段名2"=>"错误提示2"...)

前端可以根据需要自行处理。****手动验证****

3.1版本开始，可以使用validate方法实现动态和批量手动验证，例如：`\$this->validate(\$validate)->create();`其中\$validate变量的规范和_validate属性的定义规则一致，而且还可以支持函数调用（由于PHP本身的限制，在类的属性定义中不能调用函数）。

通过这一改进，以前需要支持数据自动验证，必须定义模型类的情况已经不再出现，你完全可以通过M方法实例化模型类后使用动态设置完成自动验证操作。

另外还有一个check方法，用于对单个数据的手动验证，支持部分自动验证的规则，用法如下：

```

check('验证数据','验证规则','验证类型') 验证类型支持 in between equal length regex
expire ip_allow ip_deny, 默认为regex
$model->check($value,'email');
结果返回布尔值 $model->check($value,'1,2,3','in');

```

[上一页](#) [下一页](#)

6.16 命名范围

命名范围

[上一页](#) [下一页](#)

模型命名范围功能，给模型操作提供了一系列的（连贯操作）封装，让你更方便的查询和操作数据。

定义属性

要使用命名范围功能，主要涉及到模型类的_scope属性定义和scope连贯操作方法的使用。

```
class NewsModel extends Model {
    protected $_scope = array(
        // 命名范围normal
        'normal'=>array(
            'where'=>array('status'=>1),
        ),
        // 命名范围latest
        'latest'=>array(
            'order'=>'create_time DESC',
            'limit'=>10,
        ),
    );
}
```

我们首先定义_scope属性：

_scope属性是一

个数组，每个数组项表示定义一个命名范围，命名范围的定义格式为：

'命名范围标识名'=>array('属性1'=>'值1','属性2'=>'值2'...)

命名范围标识名：可以是任意的字符串，用于标识当前定义的命名范围。

命名范围支持的属性包括：

| where | 查询条件 |

|-----|

| field | 查询字段 |

| order | 结果排序 |

| table | 查询表名 |

| limit | 结果限制 |

| page | 结果分页 |

| having | having查询 |

| group | group查询 |

| lock | 查询锁定 |

| distinct | 唯一查询 |

| cache | 查询缓存 |

每个命名范围的定义可以包括这些属性中一个或者多个。

方法调用

本文档使用 [看云](#) 构建

属性定义完成后，接下来就是使用scope方法进行命名范围的调用了，每调用一个命名范围，就相当于执行了命名范围中定义的相关操作选项。

调用某个命名范围

最简单的调用方式就直接调用某个命名范围，例如：`$Model->scope('latest')->select();` 生成的SQL语句分别是：`SELECT * FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 10`

调用多个命名范围

也可以支持同时调用多个命名范围定义，例如：

`$Model->scope('normal')->scope('latest')->select();` 或者简化为：
`$Model->scope('normal,latest')->select();` 生成的SQL都是：
`SELECT * FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 10` 如果两个命名范围的定义存在冲突，则后面调用的命名范围定义会覆盖前面的相同属性的定义。
 如果调用的命名范围标识不存在，则会忽略该命名范围，例如：
`$Model->scope('normal,new')->select();` 上面的命名范围中new是不存在的，因此只有normal命名范围生效，生成的SQL语句是：`SELECT * FROM think_news WHERE status=1`

默认命名范围

系统支持默认命名范围功能，如果你定义了一个default命名范围，例如：

```
protected $_scope = array(
    // 默认的命名范围
    'default'=>array(
        'where'=>array('status'=>1),
        'limit'=>10,
    ),
);
```

那么调用default命名范围可以直接使用：

`$Model->scope()->select();` 而无需再传入命名范围标识名
`$Model->scope('default')->select();` 虽然这两种方式是等效的。

命名范围调整

如果你需要在normal命名范围的基础上增加额外的调整，可以使用：

`$Model->scope('normal',array('limit'=>5))->select();` 生成的SQL语句是：
`SELECT * FROM think_news WHERE status=1 LIMIT 5` 当然，也可以在两个命名范围的基础上进行调整，例如：`$Model->scope('normal,latest',array('limit'=>5))->select();` 生成的SQL是：
`SELECT * FROM think_news WHERE status=1 ORDER BY create_time DESC LIMIT 5`

自定义命名范围

又或者，干脆不用任何现有的命名范围，我直接传入一个命名范围：

`$Model->scope(array('field'=>'id,title','limit'=>5,'where'=>'status=1','order`
 这样，生成的SQL变成：
`SELECT id,title FROM think_news WHERE status=1 ORDER BY create_time DESC LIM`
 本文档使用 [看云](#) 构建

与连贯操作混合使用

命名范围一样可以和之前的连贯操作混合使用，例如定义了命名范围_scope属性：

```
protected $_scope = array(
    'normal'=>array(
        'where'=>array('status'=>1),
        'field'=>'id,title',
        'limit'=>10,
    ),
);
```

然后在使用的时候，可以这样调用：

`$Model->scope('normal')->limit(8)->order('id desc')->select();` 这样，生成的SQL变成：`SELECT id,title FROM think_news WHERE status=1 ORDER BY id desc LIMIT 8` 如果定义的命名范围和连贯操作的属性有冲突，则后面调用的会覆盖前面的。

如果是这样调用：`$Model->limit(8)->scope('normal')->order('id desc')->select();` 生成的SQL则是：

`SELECT id,title FROM think_news WHERE status=1 ORDER BY id desc LIMIT 10` 命名范围功能的优势在于可以一次定义多次调用，并且在项目中也能起到分工配合的规范，避免开发人员在写CURD操作的时候出现问题，项目经理只需要合理的规划命名范围即可。

[上一页](#)[下一页](#)

6.17 自动完成

自动完成

[上一页](#)[下一页](#)

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。

填充因子格式：`array(填充字段, 填充内容, [填充条件, 附加规则])` 说明

填充 字段	必须	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 规则	必须	配合附加规则完成
填充 时间	可选	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	可选	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  
    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  
);
```

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
);
$user-> setProperty("_auto",$auto);
$user->create();
```

动态设置自动完成规则

还可以使用auto方法动态设置自动完成规则，例如：`$this->auto($auto)->create();` 其中\$auto变量的规范和_auto属性的定义规则一致，而且还可以支持函数调用（由于PHP本身的限制，在类的属性定义中不能调用函数）。

通过这一改进，以前需要支持数据自动完成，必须定义模型类的情况已经不再出现，你完全可以通过M方法实例化模型类后使用动态设置完成自动完成操作。

[上一页](#)[下一页](#)

6.18 查询语言

查询语言

[上一页](#) [下一页](#)

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，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，下面来一一讲解查询语言的内涵。

查询方式

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

一、使用字符串作为查询条件

这是最传统的方式，但是安全性不高，例如：

```
$User = M("User"); // 实例化User对象
$User->where('type=1 AND status=1')->select(); 最后生成的SQL语句是
SELECT FROM think_user WHERE type=1 AND status=1
```

二、使用数组作为查询条件

```
$User = M("User"); // 实例化User对象
$condition['name'] = 'thinkphp';
$condition['status'] = 1;
// 把查询条件传入查询方法
$User->where($condition)->select(); 最后生成的SQL语句是
SELECT FROM think_user WHERE name = 'thinkphp' AND status=1
```

如果进行多字段查询，那么字段之间的默认逻辑关系是 逻辑与 AND，但是用下面的规则可以更改默认的

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

逻辑判断，通过使用 _logic 定义查询逻辑：\$User->where(\$condition)->select(); 最后生成的SQL语句是

```
SELECT FROM think_user WHERE name = 'thinkphp' OR account = 'thinkphp'
```

三、使用对象方式来

```
$User = M("User"); // 实例化User对象
// 定义查询条件
$condition = new stdClass();
$condition->name = 'thinkphp';
$condition->status= 1;
```

查询（这里以stdClass内置对象为例）\$User->where(\$condition)->select(); 最后生成的SQL语句和上面一样

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

使用对象方式查询和使用数组查询的效果是相同的，并且是可以互换的，大多数情况下，我们建议采用数组方式更加高效，后面我们会以数组方式为例来讲解具体的查询语言用法。

表达式查询

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

| 表达式 | 含义 |

|-----|-----|

| EQ | 等于 (=) |

| NEQ | 不等于 (<>) |

| GT | 大于 (>) |

| EGT | 大于等于 (>=) |

| LT | 小于 (<) |

| ELT | 小于等于 (<=) |

| LIKE | 模糊查询 |

| [NOT] BETWEEN | (不在) 区间查询 |

| [NOT] IN | (不在) IN 查询 |

| EXP | 表达式查询，支持SQL语法 |

示例如下：

EQ：等于 (=)

例如：`$map['id'] = array('eq',100);` 和下面的查询等效 `$map['id'] = 100;` 表示的查询条件就是 `id = 100`

NEQ：不等于 (<>)

例如：`$map['id'] = array('neq',100);` 表示的查询条件就是 `id <> 100`

GT：大于 (>)

例如：`$map['id'] = array('gt',100);` 表示的查询条件就是 `id > 100`

EGT：大于等于 (>=)

例如：`$map['id'] = array('egt',100);` 表示的查询条件就是 `id >= 100`

LT：小于 (<)

例如：`$map['id'] = array('lt',100);` 表示的查询条件就是 `id < 100`

ELT：小于等于 (<=)

例如：`$map['id'] = array('elt',100);` 表示的查询条件就是 `id <= 100`

[NOT] LIKE：同sql的LIKE

例如：`$map['name'] = array('like','thinkphp%');` 查询条件就变成 `name like 'thinkphp%'` 如果配置了DB_LIKE_FIELDS参数的话，某些字段也会自动进行模糊查询。例如设置了：

`'DB_LIKE_FIELDS'=>'title|content'` 的话，使用 `$map['title'] = 'thinkphp';` 查询条

件就会变成 name like '%thinkphp%'

```
$map['a'] = array('like', array('%thinkphp%', '%tp'), 'OR');
支持数组方式, 例如 $map['b'] = array('notlike', array('%thinkphp%', '%tp'), 'AND');
生成的查询条件就是:
```

(a like '%thinkphp%' OR a like '%tp') AND (b not like '%thinkphp%' AND b not like '%tp')

[NOT] BETWEEN : 同sql的[not] between , 查询条件支持字符串或者数组, 例如:

```
$map['id'] = array('between', '1,8'); 和下面的等效:
$map['id'] = array('between', array('1', '8')); 查询条件就变成 id BETWEEN 1 AND 8
```

[NOT] IN : 同sql的[not] in , 查询条件支持字符串或者数组, 例如:

```
$map['id'] = array('not in', '1,5,8'); 和下面的等效:
$map['id'] = array('not in', array('1', '5', '8')); 查询条件就变成 id NOT IN (1,5, 8)
```

EXP : 表达式, 支持更复杂的查询情况

例如: \$map['id'] = array('in', '1,3,8'); 可以改成:

```
$map['id'] = array('exp', ' IN (1,3,8) '); exp查询的条件不会被当成字符串, 所以后面的
查询条件可以使用任何SQL支持的语法, 包括使用函数和字段名称。查询表达式不仅可用于查询条件,
也可以用于数据更新, 例如:
```

```
$User = M("User"); // 实例化User对象
// 要修改的数据对象属性赋值
$data['name'] = 'ThinkPHP';
$data['score'] = array('exp', 'score+1');// 用户的积分加1
$User->where('id=5')->save($data); // 根据条件保存修改的数据
```

快捷查询

新版增加了快捷查询方式, 可以进一步简化查询条件的写法, 例如:

```
$User = M("User"); // 实例化User对象
$map['name|title'] = 'thinkphp';
// 把查询条件传入查询方法
```

一、实现不同字段相同的查询条件 \$User->where(\$map)->select(); 查询条件就变成
name='thinkphp' OR title = 'thinkphp'

二、实现不同字段不同的查询条件

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

加在数组的最后, 表示当前是多条件匹配, 这样查询条件就变成status=1 AND title = 'thinkphp' , 查询
字段支持更多的, 例如:

```
$map['status&score&title']=array('1',array('gt','0'),'thinkphp','_multi'=>true);
查询条件就变成status=1 AND score >0 AND title = 'thinkphp'
```

注意: 快捷查询方式中 "|" 和 "&" 不能同时使用。

区间查询

ThinkPHP支持对某个字段的区间查询，例如：

```
$map['id'] = array(array('gt',1),array('lt',10)) ; 得到的查询条件是：( id > 1)
AND ( id < 10) $map['id'] = array(array('gt',3),array('lt',10), 'or') ; 得到的查
询条件是：( id > 3) OR ( id < 10)
```

```
$map['id'] = array(array('neq',6),array('gt',3), 'and'); 得到的查询条件是：( id
!= 6) AND ( id > 3)
```

最后一个可以是AND、OR或者XOR运算符，如果不写，默认是AND运算。

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

```
$map['name'] = array(array('like','%a%'), array('like','%b%'), array('like',
最后的查询条件是：( name LIKE '%a%') OR ( name LIKE '%b%') OR ( name
LIKE '%c%') OR ( name = 'ThinkPHP'))
```

组合查询

如果你需要在查询的时候同时偶尔使用字符串却又不希望丢失数组方式的灵活的话，可以考虑使用组合查询。

组合查询的主体还是采用数组方式查询，只是加入了一些特殊的查询支持，包括字符串模式查询

(`_string`)、复合查询(`_complex`)、请求字符串查询(`_query`)，混合查询中的特殊查询每次查询只能定义一个，由于采用数组的索引方式，索引相同的特殊查询会被覆盖。

一、字符串模式查询（采用`_string` 作为查询条件）

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 )
```

二、请求字符串查询方式

请求字符串查询是一种类似于URL传参的方式，可以支持简单的条件相等判断。

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

三、复合查询

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

```
$where['name'] = array('like', '%thinkphp%');
$where['title'] = array('like', '%thinkphp%');
$where['_logic'] = 'or';
$map['_complex'] = $where;
```

例如： `$map['id'] = array('gt',1);` 查询条件是


```
(id>1)AND( (namelike'%thinkphp%')OR(titlelike'%thinkphp%'))
```

复合查询使用了_complex作为子查询条件来定义，配合之前的查询方式，可以非常灵活的制定更加复杂的查询条件。

很多查询方式可以相互转换，例如上面的查询条件可以改成：

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

最后生成的SQL语句是一致的。

统计查询

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

| 方法 | 说明 |

|-----|-----|

| Count | 统计数量，参数是要统计的字段名（可选） |

| Max | 获取最大值，参数是要统计的字段名（必须） |

| Min | 获取最小值，参数是要统计的字段名（必须） |

| Avg | 获取平均值，参数是要统计的字段名（必须） |

| Sum | 获取总分，参数是要统计的字段名（必须） |

用法示例：`$User = M("User");` // 实例化User对象 获取用户数：

```
$userCount = $User->count(); 或者根据字段统计：
```

```
$userCount = $User->count("id"); 获取用户的最大积分：
```

```
$maxScore = $User->max('score'); 获取积分大于0的用户的最大积分：
```

```
$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。
```

定位查询

ThinkPHP支持定位查询，但是要求当前模型必须继承高级模型类才能使用，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();
```

SQL查询

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

1、query方法

query 执行SQL查询操作	
用法	query(\$sql,\$parse=false)
参数	query (必须) : 要查询的SQL语句 parse (可选) : 是否需要解析SQL
返回值	如果数据非法或者查询错误则返回false 否则返回查询结果数据集 (同select方法)

使用示例：

```
$Model = new Model() // 实例化一个model对象 没有对应任何数据表
```

```
$Model->query("select * from think_user where status=1");
```

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

2、execute方法

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

使用示例：

```
$Model = new Model() // 实例化一个model对象 没有对应任何数据表
```

```
$Model->execute("update think_user set name='thinkPHP' where status=1");
```

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

3、其他技巧

自动获取当前表名

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，系统还提供了的一个小的技巧来帮助解决这个问题。

```
$model = M("User");
```

例如：

```
$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。

支持连贯操作和SQL解析

新版对query和execute两个原生SQL操作方法增加第二个参数支持，表示是否需要解析SQL（默认为false 表示直接执行sql），如果设为true 则会解析SQL中的特殊字符串（需要配合连贯操作）。

```
$model->table("think_user")
    ->where(array("name"=>"thinkphp"))
    ->field("id,name,email")
```

例如，支持 如下写法：

```
->query('select %FIELD% from %TABLE% %WHERE%',true);
```

其中query方法中的%FIELD%、%TABLE%和%WHERE%字符串会自动替换为同名的连贯操作方法的解析结果SQL，支持的替换字符串包括：

替换字符串	对应连贯操作方法
%FIELD%	field
%TABLE%	table
%DISTINCT%	distinct
%WHERE%	where
%JOIN%	join
%GROUP%	group
%HAVING%	having
%ORDER%	order
%LIMIT%	limit
%UNION%	union

动态查询

借助PHP5语言的特性，ThinkPHP实现了动态查询，包括下面几种：

方法名	说明	举例
getBy	根据某个字段的值查询数据	例如，getByName,getByEmail
getFieldBy	根据某个字段查询并返回某个字段的值	例如，getFieldByName
top	获取前多少条记录（需要高级模型支持）	例如，top8，top12

一、getBy动态查询

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

```
$user = $User->getByName('liu21st');
$user = $User->getByEmail('liu21st@gmail.com');
$user = $User->getByAddress('中国深圳');
```

暂时不支持多数据字段的动态查询方法

法，请使用find方法和select方法进行查询。二、getFieldBy动态查询

针对某个字段查询并返回某个字段的值，例如

```
$user = $User->getFieldByName('liu21st','id');
```

表示根据用户的name获取用户的id值。

三、top动态查询

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（和定位查询一样，也要求当前模型类必须继承高级模型类后才能使用）。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();`

子查询

新版新增了子查询支持，有两种使用方式：

1、使用select方法

当select方法的参数为false的时候，表示不进行查询只是返回构建SQL，例如：

```
// 首先构造子查询SQL
$subQuery = $model->field('id,name')->table('tablename')->group('field')->where('field=1')
```

2、使用buildSql方法

```
$subQuery = $model->field('id,name')->table('tablename')->group('field')->where('field=1')
```

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

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

构造的子查询SQL可用于TP的连贯操作方法，例如table where等。

[上一页](#)[下一页](#)

6.19 查询锁定

查询锁定

[上一页](#)[下一页](#)

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)
```

更新锁定使用：

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

[上一页](#)[下一页](#)

6.20 字段排除

字段排除

[上一页](#)[下一页](#)

更多的情况下我们都是查询某些字段，但有些情况下面我们需要通过字段排除来更方便的查询字段，例如文章详细页，我们可能只需要排除status和updatetime字段，这样就不需要写一堆的字段名称了（有些人可能觉得为什么不用“*”查询全部字段呢，不是更方便吗，但是有一点不可否认，即使列出所有字段也比查询所有字段的效率要高哦^^），而新版的Model类的field方法可以支持排除（NOT）机制，举个例子，例如我们有一个article表，定义了有id,name,title,status,create_time,read_count,comment_count字段，当使用普通的字段查询 `$Model->field('id,name')->select();` 这是我们比较常用的查询字段方式，表示查询id,name字段。

生成的SQL语句应该是SELECT id,name FROM article

当使用下面的字段排除方式查询的时候

`$Model->field('create_time,read_count,comment_count',true);` 第二个参数表示field方法采用的是排除机制，因此实际查询的字段是除create_time,read_count,comment_count之外的其他数据表所有字段，最终要查询的字段根据实际的数据表字段有所不同。

生成的SQL语句就变成了SELECT id,name,title,status FROM article

[上一页](#)[下一页](#)

6.21 事务支持

事务支持

[上一页](#) [下一页](#)

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。

```
// 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();
}
```

例如： }

注意：系统提供的事务操作方法必须有数据库本身的支持，如果你的数据库或者数据表类型不支持事务，那么系统的事务操作是无效的。

赛赛提醒您：上文虽说必须数据库支持,也就是数据表必须支持事务处理,但是赛赛遇到一个问题就是,在进行了开启事务后两个innodb表的操作后进行了1个MyISAM表操作并返回了自增键值,但是事务回滚后连带MyISAM表的操作也回滚了。

[上一页](#) [下一页](#)

6.22 高级模型

高级模型

[上一页](#) [下一页](#)

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。

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

则是一个函数，如果设置传入整个数据对象，那么函数的参数就是整个数据对象，默认是传入数据对象中该字段的值。

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

```
protected $_filter = array(
    'content' => array( 'contentWriteFilter', 'contentReadFilter'),
)
```

其中，

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

序列化字段

序列化字段是新版推出的新功能，可以用简单的数据表字段完成复杂的表单数据存储，尤其是动态的表单数据字段。

要使用序列化字段的功能，只需要在模型中定义serializeField属性，定义格式如下：

```
protected $serializeField = array(
    'info' => array( 'name', 'email', 'address'),
);
```

Info是数据表中的实际存在的字

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

下面还是以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();
    $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;
据信息 echo $user->address;

```

查询用户数

文本字段

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

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

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

系统在查询和写入数据库的时候会自动检测文本字段，并且支持多个字段的定义。

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

只读字段

只读字段用来保护某些特殊的字段值不被更改，这个字段的值一旦写入，就无法更改。

要使用只读字段的功能，我们只需要在模型中定义readonlyField属性

```
protected $readonlyField = array('name', 'email');
```

例如，上面定义了当前模型的name和email字段为只读字段，不允许被更改。也就是说当执行save方法之前会自动过滤到只读字段的值，避免更新到数据库。

```

$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

```

下面举个例子说明下：

事实上，由于我们对name和

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

悲观锁和乐观锁

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，我们希望针对某个时间点的数据进行处理，而不希望在结算进行过程中（可能是几秒钟，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。

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
$User->where('id=3')->setInc("score",10); // 用户的积分加10
$User->where('id=3')->setInc("score",30); // 用户的积分加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

本文档使用 [看云](#) 构建

如果我们使用延迟更新方法，例如下面对用户的积分延迟更新60秒

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

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

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

效果是等效。区别在于用户数据库中的积分不是实时的。

同样，还可以使用setLazyDec进行延迟更新操作。

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getTable_name方法，而是使用getPartitionTableName方法，而且必须传入当前的数据。然后根据数据分析应该实际操作哪个数据表。因此，分表的字段值必须存在于传入的数据中，否则会进行联合查询。

返回类型

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

```
$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
```

如：\$data = \$User->returnResult(\$data, "User"); 返回自定义的User类，类的架构方

```
Class User {
    public function __construct($data){
        // 对$data数据进行处理
    }
}
```

法的参数是传入的数据。例如： }

[上一页](#) [下一页](#)

6.23 视图模型

视图模型

[上一页](#) [下一页](#)

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和用户模型的名字，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=Cat
        'User'=>array('name'=>'username', '_on'=>'Blog.user_id=User.id'),
    );
}
```

我们来解释一下定义的格式代表了什么。

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

例如：'Blog'=>array('id','name','title'); 表示BlogView视图模型要包含Blog模型中的id、name和title字段属性，这个其实很容易理解，就和数据库的视图要包含某个数据表的字段一样。而Blog相当于是给Blog模型对应的数据表定义了一个别名。

默认情况下会根据定义的名称自动获取表名，如果希望指定数据表，可以使用：

'_table'=>"test_db.test_table" 如果希望给当前数据表定义另外的别名，可以使用

'_as'=>'myBlog' BlogView视图模式除了包含Blog模型之外，还包含了Category和用户模型，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

本文档使用 [看云](#) 构建

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，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.c
    '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
```

我们可以在视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'=>'
```

视图查询

接下来，我们就可以和使用普通模型一样对视图模型进行操作了。

```
$Model = D("BlogView");
$Model->field('id,name,title,category_name,username')->where('id>10')->order(
```

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

```
$Model->field('id,name,title,category_name,username')->order('id desc')->grou
```

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");
```

本文档使用 [看云](#) 构建

```
$Model->table(
  'think_blog Blog,
  think_category Category,
  think_user User')
->field(
  'Blog.id,Blog.name,
  Blog.title,
  Category.title as category_name,
  User.name as username')
->order('Blog.id desc')
->where('Blog.category_id=Category.id AND Blog.user_id=User.id')
->select();
```

而定义了视图

模型之后，所有的字段会进行自动处理，添加表别名和字段别名，从而简化了原来视图的复杂查询。如果不使用视图模型，也可以用连贯操作的JOIN方法实现相同的功能。

[上一页](#)[下一页](#)

6.24 关联模型

关联模型

[上一页](#) [下一页](#)

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；

每个员工可以同时有多个项目组，每个项目组同时有多个员工，因此属于MANY_TO_MANY关联；

分析清楚数据表之前的关联关系后，我们才可以进行关联定义和关联操作。

关联定义

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

一个模型根据业务模型的复杂程度可以同时定义多个关联，不受限制，所有的关联定义都统一在模型类的\$_link 成员变量里面定义，并且可以支持动态定义。要支持关联操作，模型类必须继承RelationModel

```
protected $_link = array(
    '关联1' => array(
        '关联属性1' => '定义',
        '关联属性N' => '定义',
    ),
    '关联2' => array(
        '关联属性1' => '定义',
        '关联属性N' => '定义',
    ),
    '关联3' => HAS_ONE, // 快捷定义
    ...
);
```

类，关联定义的格式是：

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


```
class UserModel extends RelationModel{
    protected $_link = array(
        'Profile'=> HAS_ONE,
    );
}
```

户模型UserModel，并且添加如下关联定义： }

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 RelationModel{
    protected $_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
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。
关联查询
由于性能问题，新版取消了自动关联查询机制，而统一使用relation方法进行关联操作，relation方法不但可以启用关联还可以控制局部关联操作，实现了关联操作一切尽在掌握之中。

```
$User = D("User");
$user = $User->relation(true)->find(1); 输出$user结果可能是类似于下面的数据：
array(
    'id' => 1,
    'account' => 'ThinkPHP',
    'password' => '123456',
    'Profile' => array(
        'email' => 'liu21st@gmail.com',
        'nickname' => '流年',
    ),
)
```

我们可以看到，用户的关联数据已经被映射到数据

对象的属性里面了。其中Profile就是关联定义的mapping_name属性。

如果我们按照下面的方式定义了as_fields属性的话，

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

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

控制要关联查询的。 \$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
    $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
```

数据的查询，可以使用 \$profile = \$User->relationGet("Profile"); 事实上，除了当前的参考模型User外，其他的关联模型是不需要创建的。

关联操作

除了关联查询外，系统也支持关联数据的自动写入、更新和删除

```
$User = D("User");
$data = array();
$data["account"] = "ThinkPHP";
$data["password"] = "123456";
$data["Profile"] = array(
    'email' => 'liu21st@gmail.com',
    'nickname' => '流年',
);
```

关联写入 `$result = $User->relation(true)->add($data);` 这样就会自动写入关联的Profile数据。

同样，可以使用参数来控制要关联写入的数据：

`$result = $User->relation("Profile")->add($data);` 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);` Relation(true)会关联保存User模型定义的所有关联数据，如果只需要关联保存部分数据，可以使用：

`$result = $User->relation("Profile")->save($data);` 这样就只会同时更新关联的Profile数据。

关联保存的规则：

HAS_ONE：关联数据的更新直接赋值

HAS_MANY：的关联数据如果传入主键的值 则表示更新 否则就表示新增

MANY_TO_MANY：的数据更新是删除之前的数据后重新写入关联删除

删除用户ID为3的记录的同时删除关联数据 `$result = $User->relation(true)->delete("3");`

如果只需要关联删除部分数据，可以使用

`$result = $User->relation("Profile")->delete("3");`

[上一页](#) [下一页](#)

6.25 Mongo模型

Mongo模型

[上一页](#)[下一页](#)

Mongo模型是专门为Mongo数据库驱动而支持的Model扩展，如果需要操作Mongo数据库的话，自定义的模型类必须继承MongoModel。

Mongo模型为操作Mongo数据库提供了更方便的实用功能和查询用法，包括：

- 对MongoId对象和非对象主键的全面支持；
- 保持了动态追加字段的特性；
- 数字自增字段的支持；
- 执行SQL日志的支持；
- 字段自动检测的支持；
- 查询语言的支持；
- MongoCode执行的支持；

主键

系统很好的支持Mongo的主键类型，Mongo默认的主键名是_id，也可以通过设置pk属性改变主键名称（也许你需要用其他字段作为数据表的主键），例如：

```
Class UserModel extends MongoModel {  
    Protected $pk = 'id';  
}
```

主键支持三种类型（通过_idType属性设置），分别

是：

类型	描述
self::TYPE_OBJECT或者1 (默认类型)	采用MongoId对象，写入或者查询的时候传入数字或者字符会自动转换，获取的时候会自动转换成字符串。
self::TYPE_INT或者2	整形，支持自动增长，通过设置_autoInc 属性
self::TYPE_STRING或者3	字符串hash

```
Class UserModel extends MongoModel {  
    Protected $_idType = self::TYPE_INT;  
    protected $_autoInc = true;
```

设置主键类型示例： }

字段检测

MongoModel默认关闭字段检测，是为了保持Mongo的动态追加字段的特性，如果你的应用不需要使用Mongo动态追加字段的特性，可以设置autoCheckFields为true即可开启字段检测功能，提高安全性。一旦开启字段检测功能后，系统会自动查找当前数据表的第一条记录来获取字段列表。

如果你关闭字段检测功能的话，将不能使用查询的字段排除功能。

连贯操作

MongoModel中有部分连贯操作暂时不支持，包括：group、union、join、having、lock和distinct操作。其他连贯操作都可以很好的支持，例如：

```
$Model = new MongoModel("User");
$Model->field("name,email,age")->order("status desc")->limit("10,8")->select(
```

查询支持

Mongo数据库的查询条件和其他数据库有所区别。

首先，支持所有的普通查询和快捷查询；

表达式查询增加了一些针对MongoDb的查询用法；

统计查询目前只能支持count操作，其他的可能要自己通过MongoCode来实现了；

MongoModel的组合查询支持

_string 采用MongoCode查询

_query 和其他数据库的请求字符串查询相同

_complex MongoDb暂不支持

MongoModel提供了MongoCode方法，可以支持MongoCode方式的查询或者操作。

表达式查询

表达式查询采用下面的方式：

```
$map['字段名'] = array('表达式','查询条件');
```

因为MongoDb的特性，MongoModel的表达式查询和其他的数据库有所区别，增加了一些新的用法。

表达式不分大小写，支持的查询表达式和Mongo原生的查询语法对照如下：

查询表达式	含义	Mongo原生查询条件
neq 或者 ne	不等于	\$ne
lt	小于	\$lt
lte 或者 elt	小于等于	\$lte
gt	大于	\$gt
gte 或者 egt	大于等于	\$gte
like	模糊查询 用MongoRegex正则模拟	无
mod	取模运算	\$mod
in	in查询	\$in
nin 或者 not in	not in查询	\$nin
all	满足所有条件	\$all
between	在某个的区间	无
not between	不在某个区间	无
exists	字段是否存在	\$exists
size	限制属性大小	\$size

type	限制字段类型	\$type
regex	MongoRegex正则查询	MongoRegex实现
exp	使用MongoCode查询	无

注意，在使用like查询表达式的时候，和mysql的方式略有区别，对应关系如下：

| Mysql模糊查询 | Mongo模糊查询 |

|-----|-----|

| array('like','%thinkphp%'); | array('like','thinkphp'); |

| array('like','thinkphp%'); | array('like','^thinkphp'); |

| array('like','%thinkphp'); | array('like','thinkphp\$'); |

LIKE：同sql的LIKE

例如：`$map['name'] = array('like','^thinkphp');` 查询条件就变成 name like 'thinkphp%'

设置支持

Mongo的数据更新设置用于数据保存和写入操作，可以支持：

| 表达式 | 含义 | Mongo原生用法 |

|-----|-----|-----|

| inc | 数字字段增长或减少 | \$inc |

| set | 字段赋值 | \$set |

| unset | 删除字段值 | \$unset |

| push | 追加一个值到字段（必须是数组类型）里面去 | \$push |

| pushall | 追加多个值到字段（必须是数组类型）里面去 | \$pushall |

| addtoSet | 增加一个值到字段（必须是数组类型）内，而且只有当这个值不在数组内才增加 | \$addToSet |

|

| pop | 根据索引删除字段（必须是数组字段）中的一个值 | \$pop |

| pull | 根据值删除字段（必须是数组字段）中的一个值 | \$pull |

| pullall | 一次删除字段（必须是数组字段）中的多个值 | \$pullall |

```
$data['id'] = 5;
```

```
$data['score'] = array('inc',2);
```

例如，`$Model->save($data);`

其他

MongoModel增加了几个方法

mongoCode 执行MongoCode

getMongoNextId([字段名]) 获取自增字段的下一个ID，可用于数字主键或者其他需要自增的字段，参数为空的时候表示或者主键的。

Clear 清空当前数据表方法

[上一页](#) [下一页](#)

6.26 动态模型

动态模型

[上一页](#)[下一页](#)

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。

```
$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
要切换模型，可以使用： $User->switchModel("Adv")->top10(); // 上面的
                        $User = M("AdvModel:User"); // 实例化User对象 是基础模型类的实例
                        写法也可以改成 $User->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
    $User = M("User"); // 实例化User对象 是基础模型类的实例
    // 动态切换到关联模型类 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
使用： $user = $advUser->relation(true)->find(1);
```

[上一页](#)[下一页](#)

6.27 虚拟模型

虚拟模型

[上一页](#)[下一页](#)

有些时候，我们建立模型类但又不需要进行数据库操作，仅仅是借助模型类来封装一些业务逻辑，那么可以借助虚拟模型来完成。虚拟模型不会自动连接数据库，因此也不会自动检测数据表和字段信息，有两种方式可以定义虚拟模型：

```
Class UserModel extends Model {  
    Protected $autoCheckFields = false;
```

第一种:继承Model类 } 设置autoCheckFields属性为false后，就会关闭字段信息的自动检测，因为ThinkPHP采用的是惰性数据库连接，只要你不进行数据库查询操作，是不会连接数据库的。第二种:不继承Model类 Class UserModel { } 这种方式下面自定义模型类就是一个单纯的业务逻辑类，不能再使用模型的CURD操作方法，但是可以实例化其他的模型类进行相关操作，也可以在需要的时候直接实例化Db类进行数据库操作。

[上一页](#)[下一页](#)

6.28 多层模型支持

多层模型支持

[上一页](#)[下一页](#)

3.1版本开始，模型层（M）支持自定义分层。并且D方法，增加layer参数，具体分层的M类仍然继承Model类，用法示例：

实例化UserModel类（默认的情况）

文件位于项目的Lib/Model/UserModel.class.php `D('User');` 实例化UserLogic类 实现Logic分层

文件位于项目的Lib/Logic/UserLogic.class.php `D('User', 'Logic');` 实例化UserService类，实现Service分层

文件位于项目的Lib/Service/UserService.class.php `D('User', 'Service');` 可以配置DEFAULT_M_LAYER修改默认模型层名称（该参数默认值为Model）

[上一页](#)[下一页](#)

7. 视图

视图

[上一页](#)[下一页](#)

ThinkPHP的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，同时视图类还和模板引擎进行接口，包括完成布局渲染、输出替换、页面Trace等功能。

[上一页](#)[下一页](#)

7.1 模板定义

模板定义

[上一页](#)[下一页](#)

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/[分组名]/[模板主题]/模块名/操作名+模板后缀

模板目录默认是项目下面的Tpl，当定义分组的情况下，会按照分组名分开子目录，新版模板主题默认是空（表示不启用模板主题功能），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/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/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_FILEDEPR如果配置成 "" 的话，默认的模板文件就变成了：Tpl/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。

[上一页](#)[下一页](#)

7.2 模板赋值

模板赋值

[上一页](#)[下一页](#)

要在模板中输出变量，必须在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

```
$this->assign('name',$value);
```

//下面的写法是等效的

值，无论何种变量类型都统一使用assign赋值。 `$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['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.'']';
```

如果要或者全部的模板变量，可以调用View类的get方法支持获取全部模板变量的值，例如：

```
$this->get('name'); // 获取name模板变量的值
$this->get(); // 获取所有模板赋值变量的值
```

[上一页](#)[下一页](#)

7.3 模板输出

模板输出

[上一页](#) [下一页](#)

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()`；就可以输出模板，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('模块名:操作名')`

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');` 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。理解了这个问题，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主题的用户模块的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。

五、直接解析内容

Action类的display方法如果传入第四个参数，表示不读取模板文件而是直接解析内容。例如：

```
$this->assign('foo','ThinkPHP');  
$this->show('Hello, {$foo}!');      会在页面输出：Hello,ThinkPHP!
```

直接输出的内容仍然支持模板布局功能。

show方法也可以支持指定编码和输出格式，例如：

```
$this->show($content, 'utf-8', 'text/xml'); 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'); 如果你的网站输出编码不是默认的编码，可以使用：'DEFAULT_CHARSET'=> 'gbk' 如果要输出XML格式的，可以用：'TMPL_CONTENT_TYPE'=> 'text/xml'
```

[上一页](#) [下一页](#)

7.4 模板替换

模板替换

[上一页](#)[下一页](#)

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。模板替换适用于所有的模板引擎，包括原生的PHP模板。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/当前主题/Public/

TMPL：会替换成项目的模板目录 通常是 /项目目录/Tpl/当前主题/

（注：为了部署安全考虑，../Public和TMPL不再建议使用）

PUBLIC：会被替换成当前网站的公共目录 通常是 /Public/

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）

GROUP：会替换成当前分组的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
    '__JS__' => '/Public/JS/', // 增加新的JS类库路径替换规则
    '__UPLOAD__' => '/Uploads', // 增加新的上传路径替换规则
)
```

有了模板替换规

则后，页面上所有的PUBLIC 字符串都会被替换，那如果确实需要输出PUBLIC 字符串到模板呢，我们可以通过增加替换规则的方式，例如：

```
'TMPL_PARSE_STRING' =>array(
    '--PUBLIC--' => '__PUBLIC__', // 采用新规则输出__PUBLIC__字符串
)
```

这样增加替

换规则后，如果我们要输出PUBLIC 字符串，只需要在模板中添加--PUBLIC--，其他替换字符串的输出方式类似。

[上一页](#)[下一页](#)

7.5 获取内容

获取内容

[上一页](#)[下一页](#)

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。

[上一页](#)[下一页](#)

7.6 模板引擎

模板引擎

[上一页](#)[下一页](#)

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

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。

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

[上一页](#)[下一页](#)

7.7 布局模板

布局模板

[上一页](#)[下一页](#)

内置模板引擎提供了对布局模板功能的内置支持，如果你使用的不是内置模板引擎，可能无法使用。关于内置布局模板的功能和使用，请参考8.23的模板布局。

[上一页](#)[下一页](#)

8. 模板引擎

模板引擎

[上一页](#)[下一页](#)

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

- 支持XML标签库和普通标签的混合定义；
- 支持直接使用PHP代码书写；
- 支持文件包含；
- 支持多级标签嵌套；
- 支持布局模板功能；
- 一次编译多次运行，编译和运行效率非常高；
- 模板文件和布局模板更新，自动更新模板缓存；
- 系统变量无需赋值直接输出；
- 支持多维数组的快速输出；
- 支持模板变量的默认值；
- 支持页面代码去除Html空白；
- 支持变量组合调节器和格式化功能；
- 允许定义模板禁用函数和禁用PHP语法；
- 通过标签库方式扩展。

每个模板文件在执行过程中都会生成一个编译后的缓存文件，其实就是一个可以运行的PHP文件。模板缓存默认位于项目的Runtime/Cache目录下面，以模板文件的md5编码作为缓存文件名保存的。如果在模板标签的使用过程中发现问题，可以尝试通过查看模板缓存文件找到问题所在。

内置的模板引擎支持普通标签和XML标签方式两种标签定义，分别用于不同的目的：

| 普通标签 | 主要用于输出变量和做一些基本的操作 |

|-----|-----|

| XML标签 | 主要完成一些逻辑判断、控制和循环输出，并且可扩展 |

这种方式的结合保证了模板引擎的简洁和强大的有效融合。

[上一页](#)[下一页](#)

8.1 变量输出

变量输出

[上一页](#) [下一页](#)

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：`$this->assign('name',$name);`使用内置的模板引擎输出变量，只需要在模板文件使用：`{ $name }`模板编译后的结果就是 `<?php echo($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：`<{ $name }>`后面的内容我们都以默认的标签定义来说明。assign方法里面的第一个参数才是模板文件中使用的变量名称。如果改成下面的代码：`$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：`$this->assign('user',$user);`也就是说\$用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 3.1版本以后，类的属性输出方式有所调整，支持原生的
```

PHP对象写法，所以上面的标签需要改成：`{ $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['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
```

[上一页](#)[下一页](#)

8.2 系统变量

系统变量

[上一页](#)[下一页](#)

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。常用的系统变量输出包括下面：

| 用法 | 含义 | 例子 |

|-----|-----|-----|

| `{Think.server}` | 获取`$_SERVER` | `{Think.server.php_self}` |

| `{Think.get}` | 获取`$_GET` | `{Think.get.id}` |

| `{Think.post}` | 获取`$_POST` | `{Think.post.name}` |

| `{Think.request}` | 获取`$_REQUEST` | `{Think.request.user_id}` |

| `{Think.cookie}` | 获取`$_COOKIE` | `{Think.cookie.username}` |

| `{Think.session}` | 获取`$_SESSION` | `{Think.session.user_id}` |

| `{Think.config}` | 获取系统配置参数 | `{Think.config.app_status}` |

| `{Think.lang}` | 获取系统语言变量 | `{Think.lang.user_type}` |

| `{Think.const}` | 获取系统常量 | `{Think.const.app_name}`或`{Think.APP_NAME}` |

| `{Think.env}` | 获取环境变量 | `{Think.env.HOSTNAME}` |

| `{Think.version}` | 获取框架版本号 | `{Think.version}` |

| `{Think.now}` | 获取当前时间 | `{Think.now}` |

| `{Think.template}` | 获取当前模板 | `{Think.template}` |

| `{Think.lldelim}` | 获取模板左界定符 | `{Think.lldelim}` |

| `{Think.rdelim}` | 获取模板右界定符 | `{Think.rdelim}` |

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

`{Think.CONFIG.user.user_name}`

`SESSION`、`COOKIE`还支持二维数组的输出，例如：`{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。

```

{$_SERVER.script_name} // 输出$_SERVER变量
{$_SESSION.session_id|md5} // 输出$_SESSION变量
{$_GET.pageNumber} // 输出$_GET变量

```

以上方式还可以写成：{\$_COOKIE.name} // 输出\$_COOKIE变量 如果配置了 session和cookie前缀的话，会自动支持，例如：

```
'SESSION_PREFIX'=>'think',
```

设置参数如下：'COOKIE_PREFIX'=>'think_', 那么

{\$Think.session.name}会自动解析成 \$_SESSION['think']['name']

{\$Think.cookie.name}会自动解析成 \$_COOKIE['think_name'] 2、系统常量：使用\$Think.const 输出

```
{$Think.const.__SELF__}
```

```
{$Think.__SELF__}
```

{\$Think.const.MODULE_NAME} 或者直接使用 {\$Think.MODULE_NAME} 3、特殊变量：由

```
{$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
```

ThinkPHP系统内部定义的常量 {\$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。

[上一页](#)[下一页](#)

8.3 使用函数

使用函数

[上一页](#) [下一页](#)

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

支持变量解析缓存功能，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调用的返回结果，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，这种方式更加直接明了，而且无需通过模板变量，包括两种方式：

1、执行函数并输出返回值：

格式：`{:function(...)}`

例如，输出U函数的返回值：`{:U('User/insert')}` 编译后的PHP代码是

```
<?php echo U('User/insert'); ?>
```

2、执行函数但不输出：

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'); ?>
```

[上一页](#) [下一页](#)

8.4 默认值输出

默认值输出

[上一页](#)[下一页](#)

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"}` 默认值支持Html语法。

[上一页](#)[下一页](#)

8.5 使用运算符

使用运算符

[上一页](#)[下一页](#)

内置模板引擎包含了运算符的支持，包括对 “+” “-” “*” “/” 和 “%” 的支持，例如：

| 运算符 | 使用示例 |

| ---- | ---- |

| + | {\$a+\$b} |

| - | {\$a-\$b} |

| | {\$a\$b} |

| / | {\$a/\$b} |

| % | {\$a%\$b} |

| ++ | {\$a++} 或 {++\$a} |

| -- | {\$a--} 或 {--\$a} |

| 综合运算 | {\$a+\$b10+\$c} |

在使用运算符的时候，不再支持点语法和常规的函数用法，例如：

{ \$user.score+10 } 是错误的

{ \$user['score']+10 } 是正确的

{ \$user['score']* \$user['level'] } 正确的

{ \$user['score'] | myFun*10 } 错误的

{ \$user['score']+myFun(\$user['level']) } 正确的

[上一页](#)[下一页](#)

8.6 内置标签

内置标签

[上一页](#)[下一页](#)

变量输出使用普通标签就足够了，但是要完成其他的控制、循环和判断功能，就需要借助模板引擎的标签库功能了，系统内置标签库的所有标签无需引入标签库即可直接使用。

XML标签有两种，包括闭合标签和开放标签，一个标签在定义的时候就已经决定了是否是闭合标签还是开放标签，不可混合使用，例如：

闭合标签：`<include file="read" />` 开放标签：

`<gt name="name" value="5">value</gt>` 内置支持的标签和属性列表如下：

| 标签名 | 作用 | 包含属性 |

|-----|-----|-----|

| include | 包含外部模板文件（闭合） | file |

| import | 导入资源文件（闭合 包括js css load别名） | file,href,type,value,basepath |

| volist | 循环数组数据输出 | name,id,offset,length,key,mod |

| foreach | 数组或对象遍历输出 | name,item,key |

| for | For循环数据输出 | name,from,to,before,step |

| switch | 分支判断输出 | name |

| case | 分支判断输出（必须和switch配套使用） | value,break |

| default | 默认情况输出（闭合 必须和switch配套使用） | 无 |

| compare | 比较输出（包括eq neq lt gt egt elt heq nheq等别名） | name,value,type |

| range | 范围判断输出（包括in notin between notbetween别名） | name,value,type |

| present | 判断是否赋值 | name |

| notpresent | 判断是否尚未赋值 | name |

| empty | 判断数据是否为空 | name |

| notempty | 判断数据是否不为空 | name |

| defined | 判断常量是否定义 | name |

| notdefined | 判断常量是否未定义 | name |

| define | 常量定义（闭合） | name,value |

| assign | 变量赋值（闭合） | name,value |

| if | 条件判断输出 | condition |

| elseif | 条件判断输出（闭合 必须和if标签配套使用） | condition |

| else | 条件不成立输出（闭合 可用于其他标签） | 无 |

| php | 使用php代码 | 无 |

后面我们会详细描述每个标签的具体用法。

[上一页](#)[下一页](#)

本文档使用 [看云](#) 构建

8.7 包含文件

包含文件

[上一页](#)[下一页](#)

可以使用Include标签来包含外部的模板文件，使用方法如下：

include标签（包含外部模板文件）	
闭合	闭合标签
属性	file（必须）：要包含的模板文件，支持变量

示例：

1、使用完整文件名包含

格式：

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：

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：

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：

例如，包含blue主题的用户模块的read操作模版：`<include file="blue:User:read" />`

5、用变量控制要导入的模版

格式：

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

就可以在包含的header.html文件里面使用var1和var2变量，方法

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

注意：由于模板解析的特点，从入口模

板开始解析，如果外部模板有所更改，模板引擎并不会重新编译模板，除非在调试模式下或者缓存已经过期。如果部署模式下修改了包含的外部模板文件后，需要把模块的缓存目录清空，否则无法生效。3.1版本开始，include标签支持导入多个模板，用逗号分割即可，例如：

```
<include file='file1,file2' />
```

[上一页](#)[下一页](#)

8.8 导入文件

导入文件

[上一页](#) [下一页](#)

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/Utl/Array.js'>
<link rel="stylesheet" type="text/css" href="/App/Tpl/default/Public/css/styl
```

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标签（包含外部模板文件）|

|-----|

| 闭合 | 闭合标签 |

| 属性 | file（必须）：要包含的模板文件，支持变量 |

示例：<import type='js' file="Js.Utl.Array" /> Type属性默认是js，所以下面的效果是相同的：<import file="Js.Utl.Array" /> 还可以支持多个文件批量导入，例如：

```
<import file="Js.Utl.Array,Js.Utl.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.Utl.Array" basepath="./Common" /> 第二个是load标签，通过文件方式导入当前项目的公共JS或者CSS
```

| load标签（采用url方式引入资源文件）|

|-----|

| 闭合 | 闭合标签 |

| 属性 | href（必须）：要引入的资源文件url地址，支持变量 |

```
<load href="../Public/Js/Common.js" />
```

例如：<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" />
```

[上一页](#) [下一页](#)

8.9 Volist标签

Volist标签

[上一页](#)[下一页](#)

Volist标签主要用于在模板中循环输出数据集或者多维数组。

volist标签（循环输出数据）	
闭合	非闭合标签
属性	<div>name（必须）：要输出的数据模板变量
 id（必须）：循环变量
 offset（可选）：要输出数据的offset
 length（可选）：输出数据的长度
 key（可选）：循环的key变量，默认值为i
 mod（可选）：对key值取模，默认为2
 empty（可选）：如果数据为空显示的字符串</div>

通常模型的select方法返回的结果是一个二维数组，可以直接使用volist标签进行输出。

```
$User = M('User');  
$list = $User->select();
```

在Action中首先对模版赋值：\$this->assign('list',\$list); 在模版定义如下，循环输出用户的

```
<volist name="list" id="vo">  
    {$vo.id}  
    {$vo.name}
```

编号和姓名：</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

```
<volist name="list" id="data">  
    {$data.id}  
    {$data.name}
```

冲突，例如：</volist> 支持输出部分数据，例如输出其中的第5~15

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

条记录 </volist> 输出偶数记录

```
<volist name="list" id="vo" mod="2" >  
<eq name="mod" value="1">{$vo.name}</eq>  
</volist>
```

Mod属性还用于控制一定记录的换行，例如：

```
<volist name="list" id="vo" mod="5" >
{$vo.name}
<eq name="mod" value="4"><br/></eq>
</volist>
```

为空的时候输出提示：

```
<volist name="list" id="vo" empty="暂时没有数据" >
{$vo.id}|{$vo.name}
</volist>
```

empty属性不支持直接传入html语

法，但可以支持变量输出，例如：

```
$this->assign('empty', '<span class="empty">没有数据</span>');
$this->assign('list', $list);
```

然后在模板中使

```
<volist name="list" id="vo" empty="$empty" >
{$vo.id}|{$vo.name}
```

用：

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，

```
<volist name="list" id="vo" >
{$i}.{$vo.name}
```

例如：

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>
```

从2.1版开始允许在模板中直接使用函数设定数据集，而不需

要在控制器中给模板变量赋值传入数据集变量，如：

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

[上一页](#)[下一页](#)

8.10 Foreach标签

Foreach标签

[上一页](#)[下一页](#)

foreach标签也是用于循环输出

foreach标签（循环输出数据）	
闭合	非闭合标签
属性	name（必须）：要输出的数据模板变量 item（必须）：循环单元变量 key（可选）：循环的key变量，默认值为key

```
<foreach name="list" item="vo">
    {$vo.id}
    {$vo.name}
</foreach>
```

示例：</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。

[上一页](#)[下一页](#)

8.11 For标签

For标签

[上一页](#)[下一页](#)

For标签用于实现for循环，格式为：

for标签（循环输出数据）	
闭合	非闭合标签
属性	start（必须）：循环变量开始值 end（必须）：循环变量结束值 name（可选）：循环变量名，默认值为i step（可选）：步进值，默认值为1 comparison（可选）：判断条件，默认为lt

用法：

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

开始值、结束值、步进值和循环变量都可以支持变量，开始值和结束值是必须，其他是可选。

comparison 的默认值是lt；name的默认值是i，步进值的默认值是1，举例如下：

```
<for start="1" end="100">
{$i}
</for>
for ($i=1;$i<100;$i+=1){
    echo $i;
}
解析后的代码是
```

[上一页](#)[下一页](#)

8.12 Switch标签

Switch标签

[上一页](#)[下一页](#)

模板引擎支持Switch标签，相关的标签包括：

switch标签（分支判断输出）	
闭合	开放标签
属性	name（必须）：要输出的数据模板变量 case标签（分支判断输出）
闭合	开放标签
属性	value（必须）：变量的值，多个用“ ”分隔 break（可选）：是否要break，默认为1 default 标签
闭合	闭合标签
属性	无

```
<switch name="变量" >
  <case value="值1" break="0或1">输出内容1</case>
  <case value="值2">输出内容2</case>
  <default />默认情况
```

用法：</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

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

用“|”进行分割，例如：</switch>

表示如果

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

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

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

用： </switch>

也可以对case的value属性使用变量，

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

例如： </switch>

使用变量方式的情况下，不再支持多个条件的同时判断。

[上一页](#)[下一页](#)

8.13 比较标签

比较标签

[上一页](#)[下一页](#)

模板引擎提供了丰富的判断标签：

比较标签（判断输出数据）包括（eq,equal,notequal,neq,gt,lt,egt,elt,heq,nheq）	
闭合	非闭合标签
属性	name（必须）：变量名 value（必须）：要比较的值，支持变量

用法：<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

[上一页](#)[下一页](#)

8.14 三元运算

三元运算

[上一页](#)[下一页](#)

模板可以支持三元运算符，例如：
三元运算符中暂时不支持点语法。

```
{ $status ? '正常' : '错误' }
```

```
{ $info['status'] ? $info['msg'] : $info['error'] }
```

 注意：

[上一页](#)[下一页](#)

8.15 范围判断标签

范围判断标签

[上一页](#)[下一页](#)

Range标签用于判断某个变量是否在某个范围之内：

范围判断标签（包括innotinbetween notbetween）	
闭合	非闭合标签
属性	name（必须）：变量名 value（必须）：要比较的范围值，支持变量

用法：

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>` 如果判断不在某个范围内，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>` 可以使用between标签来判断

变量是否在某个区间范围内，可以使用：

`<between name="id"value="1,10">输出内容1</between>` 可以使用notbetween标签来判断变

量不在某个范围内：`<notbetween name="id"value="1,10">输出内容1</notbetween>` 当使用between标签的时候，value只需要一个区间范围，也就是只支持两个值，后面的值无效，例如

`<between name="id"value="1,3,10">输出内容1</between>` 实际判断的范围区间是1~3，而不是1~10，也可以支持字符串判断，例如：

`<between name="id"value="A,Z">输出内容1</between>` 所有的范围判断标签的value属性都可以使用变量，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。

[上一页](#)[下一页](#)

8.16 Present标签

Present标签

[上一页](#)[下一页](#)

可以使用present标签来判断模板变量是否已经赋值，

present标签和notpresent标签	
闭合	非闭合标签
属性	name（必须）：变量名
配合	可以结合else标签一起使用

用法：<present name="name">name已经赋值</present> 如果判断没有赋值，可以使用：
<notpresent name="name">name还没有赋值</notpresent> 可以把上面两个标签合并成为：
<present name="name">name已经赋值<else /> name还没有赋值</present>

[上一页](#)[下一页](#)

8.17 Empty标签

Empty标签

[上一页](#)[下一页](#)

可以使用empty标签判断模板变量是否为空，

empty标签和notempty标签	
闭合	非闭合标签
属性	name（必须）：变量名
配合	可以结合else标签一起使用

用法：<empty name="name">name为空值</empty> 如果判断没有赋值，可以使用：
<notempty name="name">name不为空</notempty> 可以把上面两个标签合并成为：
<empty name="name">name为空<else /> name不为空</empty>

[上一页](#)[下一页](#)

8.18 Defined标签

Defined标签

[上一页](#)[下一页](#)

可以使用defined标签判断常量是否已经有定义：

defined标签和notdefined标签	
闭合	非闭合标签
属性	name（必须）：变量名

用法：<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>

[上一页](#)[下一页](#)

8.19 Define标签

Define标签

[上一页](#)[下一页](#)

可以使用define标签进行常量定义：

defined标签和notdefined标签	
闭合	闭合标签
属性	name（必须）：常量名 value（必须）：常量值，支持变量
配合	可以结合else标签一起使用

用法：<define name="MY_DEFINE_NAME" value="3"/> 在运行模板的时候 定义了一个MY_DEFINE_NAME的常量。

[上一页](#)[下一页](#)

8.20 Assign标签

Assign标签

[上一页](#)[下一页](#)

可以使用assign标签进行赋值：

assign标签（在模板中给变量赋值）	
闭合	闭合标签
属性	name（必须）：模板变量名 value（必须）：变量值，支持变量

用法示例：`<assign name="var" value="123" />` 在运行模板的时候 赋值了一个var的变量，值是123。

[上一页](#)[下一页](#)

8.21 IF标签

IF标签

[上一页](#)[下一页](#)

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。

If标签（条件判断标签）	
闭合	非闭合标签
属性	condition（必须）：要判断的条件
elseif标签（条件判断标签）	
闭合	闭合标签
属性	condition（必须）：要判断的条件
else标签（条件判断标签）	
闭合	闭合标签
属性	无

用法示例：

```
<if condition="($name eq 1) OR ($name gt 100) "> value1
<elseif condition="$name eq 2"/>value2
<else /> value3
</if>
```

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

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

必须改成：

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

除此之外，我们可以在condition属性里面使用php代码，例如：

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

condition属性可以支持点语法和对象语法，例如：

自动判断user变量是数组还是对象

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

或者知道user变量是对象

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

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

[上一页](#)[下一页](#)

8.22 标签嵌套

标签嵌套

[上一页](#)[下一页](#)

模板引擎支持标签的多层嵌套功能，可以对标签库的标签指定可以嵌套。

系统内置的标签中，`volist`、`switch`、`if`、`elseif`、`else`、`foreach`、`compare`（包括所有的比较标签）、`(not) present`、`(not) empty`、`(not) defined`等标签都可以嵌套使用。例如：

```
<volist name="list" id="vo">
    <volist name="vo['sub']" id="sub">
        {$sub.name}
    </volist>
</volist>
```

上面的标签可以用于输出双重循环。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，例如：`'TAG_NESTED_LEVEL' => 5` 可以改变循环嵌套级别为5级。

[上一页](#)[下一页](#)

8.23 使用PHP代码

使用PHP代码

[上一页](#)[下一页](#)

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标签（在模板中使用php代码）	
闭合	非闭合标签
属性	无

例如：`<php>echo 'Hello,world!';</php>` 我们建议需要使用PHP代码的时候尽量采用php标签，因为原生的PHP语法可能会被配置禁用而导致解析错误。

第二种就是直接使用原始的php代码：`<?php echo 'Hello,world!'; ?>` 注意：php标签或者php代码里面就不能再使用标签（包括普通标签和XML标签）了，因此下面的几种方式都是无效的：

`<php><eq name='name' value='value'>value</eq></php>` Php标签里面使用了eq标签，因此无效 `<php>if({$user} != 'ThinkPHP') echo 'ThinkPHP' ;</php>` Php标签里面使用了{\$user}普通标签输出变量，因此无效。

`<php>if($user.name != 'ThinkPHP') echo 'ThinkPHP' ;</php>` Php标签里面使用了\$user.name 点语法变量输出，因此无效。

简而言之，在PHP标签里面不能再使用PHP本身不支持的代码。

如果设置了TMPL_DENY_PHP参数为true，就不能在模板中使用原生的PHP代码，但是仍然支持PHP标签输出。

[上一页](#)[下一页](#)

8.24 模板布局

模板布局

[上一页](#) [下一页](#)

新版模板引擎内置了布局模板功能支持，可以方便的实现模板布局以及布局嵌套功能。有两种布局模板的支持方式：

第一种方式是 以布局模板为入口的方式

该方式需要配置开启LAYOUT_ON 参数（默认不开启），并且设置布局入口文件名LAYOUT_NAME（默认为layout）。

开启LAYOUT_ON后，我们的模板渲染流程就有所变化，例如：

```
Class UserAction extends Action {
    Public function add() {
        $this->display('add');
    }
}
```

在不开启LAYOUT_ON布局模板之前，会直接渲染

Tpl/User/add.html 模板文件,开启之后，首先会渲染Tpl/layout.html 模板，布局模板的写法和其他模板的写法类似，本身也可以支持所有的模板标签以及包含文件，区别在于有一个特定的输出替换变量{CONTENT}，例如，下面是一个典型的layout.html模板的写法：{__CONTENT__} 读取layout模板之后，会再解析User/add.html 模板文件，并把解析后的内容替换到layout布局模板文件的{CONTENT} 特定字符串。

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

如果项目需要使用不同的布局模板，可以动态的配置LAYOUT_NAME参数实现。

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

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

第二种方式是以当前输出模板为入口的方式

以前面的输出模板为例，这种方式的入口还是在User/add.html 模板，但是我们可以修改下add模板文件的内容，在头部增加下面的布局标签：<layout name="layout" /> 表示当前模板文件需要使用layout.html 布局模板文件，而布局模板文件的写法和上面第一种方式是一样的。当渲染User/add.html 模板文件的时候，如果读取到layout标签，则会把当前模板的解析内容替换到layout布局模板的{CONTENT} 特定字符串。

如果需要使用其他的布局模板，可以改变layout的name属性，例如：

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

```
<include file="Public:header" />
<div id="main" class="main" >
{__CONTENT__}
</div>
```

`<include file="Public:bottom" />` 在引入的header和footer模板文件中也可以添加layout标签，例如header模板文件的开头添加如下标签：`<layout name="menu" />` 这样就实现了在头部模板中引用了menu布局模板。

也可以采用两种布局方式的结合，可以实现更加复杂的模板布局以及嵌套功能。

[上一页](#)[下一页](#)

8.25 模板继承

模板继承

[上一页](#) [下一页](#)

模板继承是3.1.2版本添加的一项更加灵活的模板布局方式，模板继承不同于模板布局，甚至来说，应该在模板布局的上层。模板继承其实并不难理解，就好比类的继承一样，模板也可以定义一个基础模板（或者是布局），并且其中定义相关的区块（block），然后继承（extend）该基础模板的子模板中就可以对基础模板中定义的区块进行重载。

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

每个区块由标签组成，并且不支持block标签的嵌套。

下面就是基础模板中的一个典型的区块设计（用于设计网站标题）：

`<block name="title"><title>网站标题</title></block>` block标签必须指定name属性来标识当前区块的名称，这个标识在当前模板中应该是唯一的，block标签中可以包含任何模板内容，包括其他标签和变量，例如：`<block name="title"><title>{$web_title}</title></block>` 你甚至还可以在区块中加载外部文件：

`<block name="include"><include file="Public:header" /></block>` 一个模板中可以定义任意多个名称标识不重复的区块，例如下面定义了一个base.html基础模板：

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

然后我们

在子模板（其实是当前操作的入口模板）中使用继承：

```

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

```

可以看到，子模板中使用了

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

```
<extend name="Public:base" /> 或者使用绝对文件路径加载
```

`<extend name="./Tpl/Public/base.html" />` 在当前子模板中，只能定义区块而不能定义其他的模板内容，否则将会直接忽略，并且只能定义基础模板中已经定义的区块。

```

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

```

例如，如果采用下面的定义：

导航部分将是无效的，不会显示在模板中。

在子模板中，可以对基础模板中的区块进行重载定义，如果没有重新定义的话，则表示沿用基础模板中的区块定义，如果定义了一个空的区块，则表示删除基础模板中的该区块内容。

上面的例子，我们就把left区块的内容删除了，其他的区块都进行了重载。

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

[上一页](#)[下一页](#)

8.26 原样输出

原样输出

[上一页](#)[下一页](#)

literal标签（保持原样输出）	
闭合	非闭合标签
属性	无

可以使用literal标签来防止模板标签被解析，例如：

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

上面的if标签被literal标签包含，因此if标签

里面的内容并不会被模板引擎解析，而是保持原样输出。

如果你的php标签中需要输出类似{\$user} 或者 XML标签的情况，可以通过添加literal标签解决混淆问题，例如：`<php>echo '{$Think.config.CUSTOM.'.$key.'}';</php>` 这个php标签中的{\$Think 可能会被模板引擎误当做标签解析，解决的办法就是加上literal，例如：

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

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

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

[上一页](#)[下一页](#)

8.27 模板注释

模板注释

[上一页](#)[下一页](#)

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。

```
{// 这是模板注释内容 }
```

```
{/* 这是模板
```

例如：`注释内容*/ }` 模板注释支持多行，模板注释在生成编译缓存文件后会自动删除，这一点和Html的注释不同。

[上一页](#)[下一页](#)

8.28 引入标签库

引入标签库

[上一页](#) [下一页](#)

前面我们所讲述的标签用法都是内置的标签库的用法，事实上，内置模板引擎的标签库是可以扩展和增加标签的，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：

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）。

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' />` 和 `<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。

[上一页](#) [下一页](#)

8.29 修改定界符

修改定界符

[上一页](#)[下一页](#)

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

要更改普通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'=>'<', 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和模板注释。如果要使用其它功能，请使用XML模板标签。XML模板标签可以用于模板变量输出、文件包含、条件控制、循环输出等功能，而且完全可以自己扩展功能。如果你觉得XML标签无法在正在使用的编辑器里面无法编辑，还可以更改XML标签库的起始和结束标签，请修

改下面的配置参数： TAGLIB_BEGIN //标签库标签开始标签
TAGLIB_END //标签库标签结束标记 例如在项目配置文件中增加下面

'TAGLIB_BEGIN'=>'[' ,
的配置： 'TAGLIB_END'=>']' , 原来的

相等不相等

就必须改成 [eq name="name" value="value"]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。如果你定制了普通表情的定界符，而且默认跳转页面用的是系统默认的话，记得修改下默认跳转模板中的变量定界符。

[上一页](#)[下一页](#)

8.30 避免JS混淆

避免JS混淆

[上一页](#)[下一页](#)

如果使用内置的模板引擎，而且采用默认的标签设置的话，在某些情况下，如果不注意，`{$('name').value}` 这样的JS代码很容易被内置模板引擎误解析。

有三个方法可以解决类似的混淆问题：

1、`{$('name').value}`改成`{ $('name').value}`

因为内置模板引擎的解析规则是"`{`"后面紧跟"`$`"符号才会解析变量 因此只要在"`{`" 和"`$`"之间添加空格就不会被误解析了

2、使用内置的literal标签包含JS代码

JS代码 包含在literal标签中的代码将会直接输出，不进行任何解析

3、定制模板引擎标签的定界符

例如：`'TMPL_L_DELIM'=>'<{'`,`'TMPL_R_DELIM'=>'>'}`这样就和JS代码区别开来了。

[上一页](#)[下一页](#)

9. 日志

日志

[上一页](#)[下一页](#)

日志的处理工作是由系统自动进行的，在开启日志记录的情况下，会记录下允许的日志级别的所有日志信息。其中，为了性能考虑，SQL日志级别必须在调试模式开启下有效，否则就不会记录。系统的日志记录由核心的Log类完成，提供了多种方式记录了不同的级别的日志信息。

[上一页](#)[下一页](#)

9.1 日志级别

日志级别

[上一页](#)[下一页](#)

ThinkPHP对系统的日志按照级别来分类，包括：

| EMERG | 严重错误，导致系统崩溃无法使用 |

|-----|-----|

| ALERT | 警戒性错误，必须被立即修改的错误 |

| CRIT | 临界值错误，超过临界值的错误 |

| ERR | 一般性错误 |

| WARN | 警告性错误，需要发出警告的错误 |

| NOTICE | 通知，程序可以运行但是还不够完美的错误 |

| INFO | 信息，程序输出信息 |

| DEBUG | 调试，用于调试信息 |

| SQL | SQL语句，该级别只在调试模式开启时有效 |

要开启日志记录，必须在配置中开启LOG_RECORD参数，以及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_LEVEL'   => 'EMERG,ALERT,CRIT,ERR', // 只记录EMERG ALERT CRIT ERR 错误
```

[上一页](#)[下一页](#)

9.2 记录方式

记录方式

[上一页](#) [下一页](#)

日志的记录方式包括下面四种方式：

| 记录方式 | 说明 | 常量标识 |

|-----|-----|-----|

| SYSTEM | 日志发送到PHP的系统日志记录 | 0 |

| MAIL | 日志通过邮件方式发送 | 1 |

| FILE | 日志通过文件方式记录（默认方式） | 3 |

| SAPI | 日志通过SAPI方式记录 | 4 |

日志的记录格式：记录时间 | 访问URL | 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：

```
[2012-01-15T18:09:22+08:00] /Index/index|NOTIC: [8] Undefined variable: verif
[2012-01-15T18:09:22+08:00] /Index/index | SQL: RunTime:0.214238s SQL = SHOW
[2012-01-15T18:09:22+08:00] /Index/index | SQL: RunTime:0.039159s SQL = SELE
think_user WHERE ( account = 'admin' ) AND ( status > 0 ) LIMIT 1 默认采用文件方
式记录日志信息，日志文件的命名格式是：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

如果需要使用其他方式记录日志，可以设置LOG_TYPE参数，例如下面设置了采用邮件方式发送日志记

```
'LOG_TYPE' =>1, // 采用邮件方式记录日志
```

```
'LOG_DEST' =>'admin@domain.com', // 要发送日志的邮箱
```

录：'LOG_EXTRA' =>'From: webmaster@example.com', // 邮件的发件人设置 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。3.1版本开始，简化了日志记录的信息，减少日志文件的大小，包括：

- 1、去掉了每条日志记录的请求地址，改为放到每次访问日志保存的开头；
- 2、去掉重复的日志时间显示，改为记录到每次请求的开头；
- 3、在日志头部添加了请求的IP地址信息。

[上一页](#) [下一页](#)

9.3 手动记录

手动记录

[上一页](#)[下一页](#)

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 方法

Log::write 直接写入日志	
用法	Log::write(\$message,\$level=self::ERR,\$type="",\$destination="",\$extra="")
参数	message（必须）：要记录的日志信息，字符串 level（可选）：要记录的日志级别，默认为ERR 错误 type（可选）：日志记录方式，默认为空取LOG_TYPE配置 destination（可选）：日志记录目标，默认为空自动生成或LOG_DEST配置 extra（可选）：日志记录额外参数，默认为空取LOG_EXTRA配置
返回值	无

使用示例：`Log::write('调试的SQL:'.$SQL, Log::SQL);`表示用默认的日志记录方式记录调试SQL信息

二、使用Log::record和 Log::save方法

Log::record记录日志	
用法	Log::record(\$message,\$level=self::ERR,\$record=false)
参数	message（必须）：要记录的日志信息，字符串 level（可选）：要记录的日志级别，默认为ERR 错误 record（可选）：是否强制记录，默认为false表示判断LOG_LEVEL配置
返回值	无

Log::record方法必须结合Log::save方法才能完成日志记录，因为record方法只是把日志信息保存到内存，并没有真正写入日志，直到调用Log::save方法。

Log::save 保存记录的日志	
用法	Log::save(\$type="",\$destination="",\$extra="")

参数	type（可选）：日志记录方式，默认为空取LOG_TYPE配置 destination（可选）：日志记录目标，默认为空自动生成或LOG_DEST配置 extra（可选）：日志记录额外参数，默认为空取LOG_EXTRA配置
返回值	无

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

[上一页](#)[下一页](#)

10. 错误

错误

[上一页](#)[下一页](#)

[上一页](#)[下一页](#)

10.1 异常处理

异常处理

[上一页](#)[下一页](#)

和PHP默认的异常处理不同，ThinkPHP抛出的不是单纯的错误信息，而是一个人性化的错误页面，如下图所示：



ThinkPHP 3.0RC1 { Fast & Simple OOP PHP Framework } -- [WE CAN DO IT JUST THINK]

只有在调试模式下面才能显示具体的错误信息，如果在部署模式下面，你可能看到的是一个统一错误的提示文字，如果你试图在部署模式下访问一个不存在的模块或者操作，会发送404错误。调试模式下面一旦系统发生严重错误会自动抛出异常，也可以用ThinkPHP定义的throw_exception方法手动抛出异常。

throw_exception 抛出异常	
用法	throw_exception(\$msg, \$type='ThinkException', \$code=0)
参数	msg（必须）：要抛出的异常信息，字符串 type（可选）：异常类型，默认为ThinkException，如果异常类型不存在，则会调用系统的halt方法直接输出错误信息。 code（可选）：异常代码，默认为0
返回值	无

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。

[上一页](#)[下一页](#)

10.2 异常模板

异常模板

[上一页](#)[下一页](#)

系统内置的异常模板在系统目录的Tpl/think_exception.tpl，可以通过修改系统模板来修改异常页面的显示。也通过设置TMPL_EXCEPTION_FILE 配置参数来修改系统默认的异常模板文件，例如：

```
'TMPL_EXCEPTION_FILE' => APP_PATH.'/Public/exception.tpl'
```

异常模板中可以使用的异常变量有：

`$e['file']` 异常文件名

`$e['line']` 异常发生的文件行数

`$e['message']` 异常信息

`$e['trace']` 异常的详细Trace信息

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

[上一页](#)[下一页](#)

10.3 异常显示

异常显示

[上一页](#)[下一页](#)

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。新版如果关闭调试模式的话，为了安全起见，默认就是关闭异常信息提示。

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

`'ERROR_PAGE' =>'/Public/error.html'` 如果不在当前域名，还可以指定域名：

`'ERROR_PAGE' =>'http://www.myDomain.com/Public/error.html'` 注意ERROR_PAGE所指向的页面不能再使用异常的模板变量了。

[上一页](#)[下一页](#)

11. 调试

调试

[上一页](#)[下一页](#)

这里所说的调试并非是指调试模式，ThinkPHP提供给开发人员很多的调试手段和方法，而配合调试模式则是可以更加方便调试工作。

[上一页](#)[下一页](#)

11.1 运行状态

运行状态

[上一页](#)[下一页](#)

我们可以配置SHOW_RUN_TIME参数开启当前页面的运行状态显示，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:0.2463s (Load:0.0003s Init:0.0010s Exec:0.1095s Template:0.1355s )|DE
```

表示的含义是：

运行信息：整体执行时间0.2463s (加载:0.0003s 初始化:0.0010s 执行:0.1095s 模板:0.1355s)

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，否则也不会显示。

如果开启上面的运行状态显示，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
'SHOW_LOAD_FILE'     => true, // 显示加载文件数
'SHOW_FUN_TIMES'     => true, // 显示函数调用次数
```

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
```

默认的情况下，运行时间的显示是在Html页面的最后，如果需要在制定位置显示，只需要在Html模板文件中相关位置加上 { RUNTIME} 即可，系统在输出页面的时候会自动在该位置替换运行时间的信息显示。

注意：新版即使在调试模式下面，也不会自动开启运行时间显示，需要手动开启。

[上一页](#)[下一页](#)

11.2 页面Trace

页面Trace

[上一页](#) [下一页](#)

页面Trace功能是ThinkPHP提供给开发人员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，并支持自定义显示。

页面Trace功能无论是调试模式还是部署模式都有效，要开启页面Trace功能，需要在项目配置文件中设置：`'SHOW_PAGE_TRACE' => true`，// 显示页面Trace信息 系统默认的Trace信息包括：请求时间、当前页面、请求协议、运行信息、会话ID、日志记录和文件加载情况。默认的页面Trace的显示如图所示：

```

页面Trace信息
请求时间 : 2012-01-21 20:59:19
当前页面 : /App/Examples/Blog/Blog/category/id/1
请求协议 : HTTP/1.1 GET
运行信息 : Process: 0.0643s ( Load:0.0011s Init:0.0014s Exec:0.0260s Template:0.0357s ) | DB :10 queries 0
writes | UseMem:1,125 kb | LoadFile:21 | CallFun:63,1370
会话ID : pl77ggdecram04scvgufi5s2h3
日志记录 : 无日志记录
加载文件 : 21
[0] => E:\www\App\Examples\Blog\index.php
[1] => E:\www\App\ThinkPHP\ThinkPHP.php
[2] => ./Runtime/~runtime.php
[3] => E:\www\App\Examples\Blog\Runtime/~runtime.php
[4] => E:\www\App\ThinkPHP\Extend\Behavior\CheckLangBehavior.class.php
[5] => E:\www\App\Examples\Blog\Lib\Action\BlogAction.class.php
[6] => E:\www\App\Examples\Blog\Lib\Action\PublicAction.class.php
[7] => E:\www\App\ThinkPHP\Lib\Core\Model.class.php
[8] => E:\www\App\ThinkPHP\Lib\Core\Db.class.php

```

新版的页面Trace信息显示中已经包含了运行状态时间显示，所以开启页面Trace功能后无需再开启运行时间显示了。我们的建议是运行时间显示功能用于部署模式需要的时候开启。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/page_trace.tpl，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
    $_trace = trace();
    foreach ($_trace as $key=>$info){
        echo $key.' : '.$info.'  
';
    }

```

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合并到系统默认的Trace信息，所以不需要再添加系统默认的页面trace信息，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

```
        trace('执行时间', $runTime);
        trace('Name的值', $name);
以用于系统的开发阶段调试。例如： trace('GET变量', dump($_GET, false)); trace方法支持批量
        $info['执行时间'] = $runTime;
        $info['Name的值'] = $name;
设置，例如： $info['GET变量'] = dump($_GET, false); trace($info); 这种方式的trace信息显示在
页面Trace信息的最开始。
```

3.1版本的页面Trace

3.1版本对页面Trace功能进行了增强，更加方便开发过程中的调试，并接管了一部分日志功能。

使用

页面Trace功能是ThinkPHP框架为开发人员精心设计的一个方便调试的内置行为扩展工具，经历了多个版本的改进后，由开始的具备简单页面信息到现在的全面调试支持。

页面Trace功能分为两个层面：一、是页面Trace的显示界面；二、是提供调试支持的trace方法。

要开启页面Trace显示界面，需要开启SHOW_PAGE_TRACE参数： 'SHOW_PAGE_TRACE'=>true 该配置参数默认关闭。开启后，页面的右下角会出现TP的logo，



点击即可打开页面Trace窗口。

基本 文件 流程 错误 SQL 调试

请求信息：2012-08-31 17:17:13 HTTP/1.1 GET : /examples/Trace/

运行时间：0.0589s (Load:0.0051s Init:0.0109s Exec:0.0280s Template:0.0149s)

内存开销：2,668.05 kb

查询信息：4 queries 0 writes

文件加载：35

缓存信息：0 gets 0 writes

配置加载：127

会话信息：SESSION_ID=9ahqvq1015ehtd78n5ucg9ogg6

页面Trace框架有6个选项卡，分别是基本、文件、流程、错误、SQL和调试，点击不同的选项卡会切换到不同的Trace信息窗口。

基本：当前页面的基本摘要信息，例如执行时间、内存开销、文件加载数、查询次数等等。

文件：详细列出当前页面执行过程中加载的文件及其大小。

流程：会列出当前页面执行到的行为和相关流程（待完善）。

错误：当前页面执行过程中的一些错误信息，包括警告错误。

SQL：当前页面执行到的SQL语句信息。

调试：开发人员在程序中进行的调试输出。

要在调试选项卡中显示调试信息，则是通过trace方法，该方法可以加到应用程序的任意位置，调用格式如下：

```
trace('调试变量','显示标签')
```

例如，

```
trace($user,'用户信息');
```

\$user变量可能是一个用户信息数组，那么该变量的值就会显示到页面Trace窗口的调试选项卡中。

页面Trace的选项卡是可以定制和扩展的，默认的配置为：

```
'TRACE_PAGE_TABS'=>array('base'=>'基本','file'=>'文件','think'=>'流程','error'=
```

也就是我们看到的默认情况下显示的选项卡，如果你希望增加新的选项卡：用户，则可以修改配置如下：

```
'TRACE_PAGE_TABS'=>array('base'=>'基本','file'=>'文件','think'=>'流程','error'=
```

我们把刚才的用户信息调试输出到用户选项卡，trace方法的用法修改如下：

```
trace($user,'用户信息','user');
```

第三个参数表示选项卡的标识，和我们在TRACE_PAGE_TABS中配置的对应。

默认情况下，页面Trace窗口显示的信息是不会保存的，如果希望保存这些trace信息，我们可以配置PAGE_TRACE_SAVE参数：'PAGE_TRACE_SAVE'=>true 开启页面trace信息保存后，每次的页面Trace信息会以日志形式保存到项目的日志目录中，命名格式是：

当前日期_trace.log

例如：12-06-21_trace.log 如果不希望保存所有的选项卡的信息，可以设置需要保存的选项卡，例如：'PAGE_TRACE_SAVE'=>array('base','file','sql'); 设置后只会保存base、file和sql三个选项卡的信息。

3.1版本的trace方法不支持批量赋值操作。

[上一页](#) [下一页](#)

11.3 调试方法

调试方法

[上一页](#)[下一页](#)

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。

变量调试

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

dump 浏览器友好的变量输出	
用法	dump(\$var, \$echo=true, \$label=null, \$strict=true)
参数	var（必须）：要输出的变量，支持所有变量类型 echo（可选）：是否直接输出，默认为true，如果为false则返回但不输出 label（可选）：变量输出的label标识，默认为空 strict（可选）：输出变量类型，默认为true，如果为false则采用print_r输出
返回值	如果echo参数为false 则返回要输出的字符串

```
$Blog = D("Blog");
$bblog = $Blog->find(3);
```

使用示例：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) ""
}
```

性能调试

开发过程中，有些时候为了测试性能，经常需要调试某段代码的运行时间或者内存占用开销，系统提供了

一些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 区间调试开始（记录初始时间和内存占用）	
用法	debug_start(\$label=)
参数	label（可选）：区间的label标识，默认为空
返回值	无

debug_end 区间调试结束（记录区间结束时间和内存占用 并输出结果）	
用法	debug_end(\$label=)
参数	label（可选）：区间的label标识，默认为空， 必须和debug_start的label对应才能输出正确的区间结果
返回值	无

注意：debug_start和debug_end 方法中的内存占用输出需要环境支持memory_get_usage方法，否则只会显示时间信息。

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

使用示例： debug_end('run'); 会输出下面的运行信息：

Process run: Times 0.007730s Memories 76 k 如果仅仅需要调试时间开销，还可以使用内置的G函数来更方便实现

G 用于记录和统计时间（微秒）	
用法	G(\$start,\$end=,\$dec=4)
参数	start（必须）：起始位置标识 end（可选）：记录结束标记并统计时间 dec（可选）：调试时间的统计精度，默认为小数点后4位
返回值	如果end为空或者是一个浮点数，无返回值。 如果end是一个字符串，则返回从start到end位置的使用时间。

```
G('run');
$log = D("Blog");
$log->select();
```

使用示例： echo G('run','end').'s'; 除了上面的 函数外，系统还提供了一个扩展调试类Debug

[table][table]

Debug::mark 标记调试位（并记录该位置的时间和内存占用）	
用法	Debug::mark(\$name)
参数	name（必须）：调试标记位的name标识

返回值	无
Debug::useTime 统计区间标记调试位的使用时间	
用法	useTime(\$start,\$end,\$decimals = 6)
参数	start（必须）：调试开始位置标识 end（必须）：调试结束位置标识 decimals（精度）：调试时间的统计精度默认为小数点后6位
返回值	区间位置的使用时间（字符串）
Debug::useMemory 统计区间标记调试位的内存占用	
用法	useMemory(\$start,\$end)
参数	start（必须）：调试开始位置标识 end（必须）：调试结束位置标识
返回值	区间位置的内存占用（字符串）
Debug::getMemPeak 统计区间标记调试位的内存占用峰值	
用法	getMemPeak (\$start,\$end)
参数	start（必须）：调试开始位置标识 end（必须）：调试结束位置标识
返回值	区间位置的内存占用峰值（字符串）

要使用Debug类调试的话，首先需要导入Debug类，Debug类位于扩展目录下面的Library/ORG/Util/Debug.class.php，所以首先要导入：

```
import('ORG.Util.Debug');
Debug::mark('run');
$log = D("Blog");
$log->select();
Debug::mark('end');
echo Debug::useTime('run','end'). 's';
echo Debug::useMemory('run','end'). 'kb';
```

断点调试

凭借强大的页面Trace信息功能支持，ThinkPHP可以支持断点调试功能。

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

我们只需要在不同的位置对某个变量进行trace输出即可，例如：

错误调试

如果需要我们可以使用下面的方法输出错误信息并中断执行：

halt(\$msg)//输出错误信息，并中止执行

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

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

来输出上次执行的sql语句。例如：echo \$User->getLastSql(); 输出结果是

SELECT FROM think_user WHERE id = 1

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

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

后SQL记录。 输出结果是

SELECT FROM think_info WHERE id = 2

SELECT FROM think_user WHERE id = 1

SELECT FROM think_info WHERE id = 2

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

注意：Mongo数据库驱动由于接口的特殊性，不存在执行SQL的概念，因此SQL日志记录功能是额外封装实现的，所以出于性能考虑，只有在开启调试模式的时候才支持使用getLastSql方法获取最后执行的SQL记录。

[上一页](#)[下一页](#)

12. 缓存

缓存

[上一页](#)[下一页](#)

ThinkPHP提供了方便的缓存方式，包括数据缓存、静态缓存和查询缓存等，支持包括文件方式、APC、Db、Memcache、Shmop、Sqlite、Redis、Eaccelerator和Xcache在内的动态数据缓存类型，以及可定制的静态缓存规则，并提供了快捷方法进行存取操作。

[上一页](#)[下一页](#)

12.1 缓存方式

缓存方式

[上一页](#)[下一页](#)

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、Redis和Xcache。

[上一页](#)[下一页](#)

12.2 动态缓存

动态缓存

[上一页](#)[下一页](#)

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，首先实例化缓存类：

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

缓存方式	可以支持File、Apachenote、Apc、Eaccelerator、Memcache、Shmop、Sqlite、Db、Redis和Xcache	
缓存参数 (根据不同的缓存方式存在不同的参数)	通用缓存参数	expire 缓存有效期 (默认由DATA_CACHE_TIME参数配置) length 缓存队列长度 (默认为0) queue 缓存队列方式 (默认为file 还支持xcache和apc)
	缓存方式	额外支持的缓存参数
	File (文件缓存)	temp 缓存目录 (默认由DATA_CACHE_PATH参数配置)
	Apachenote 缓存	host 缓存服务器地址 (默认为127.0.0.1)
	Apc缓存	暂无其他参数
	Eaccelerator 缓存	暂无其他参数
	Xcache缓存	暂无其他参数
	Memcache	host 缓存服务器地址 (默认为127.0.0.1) port 端口 (默认为MEMCACHE_PORT参数或者11211) timeout 缓存超时 (默认由DATA_CACHE_TIME参数设置) persistent 长连接 (默认为false)
	Shmop	size (默认由SHARE_MEM_SIZE参数设置) tmp (默认为TEMP_PATH) project (默认为s) length 缓存队列长度 (默认为0)
	Sqlite	db 数据库名称 (默认:memory:) table 表名 (默认为sharedmemory) persistent 长连接 (默认为false)
	Db	db 数据库名称 (默认由DB_NAME参数配置) table 数据表名称 (默认由DATA_CACHE_TABLE参数配置)
	Redis	host 服务器地址 (默认由REDIS_HOST参数配置或者127.0.0.1) port端口 (默认由REDIS_PORT参数配置或者6379) timeout 超时时间 (默认由DATA_CACHE_TIME配置或者false) persistent

		长连接（默认为false）
--	--	---------------

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'));
```

设置缓存参数

实例化缓存类的时候如果没有指定缓存参数，可以通过setOptions方法具体指定：

```
$Cache->setOptions('temp', 'ThinkPHP'); 具体缓存参数根据不同的缓存方式有所区别。
```

如果需要获取当前缓存驱动的参数，可以使用：`$value = $Cache->getOptions('temp');`

```

$Cache->set('name', 'ThinkPHP'); // 缓存name数据
$value = $Cache->get('name'); // 获取缓存的name数据
存取缓存数据 $Cache->rm('name'); // 删除缓存的name数据 或者使用下面的方
$Cache->name = 'ThinkPHP';
$value = $Cache->name;
法是等效的： Unset($Cache->name); 缓存设置方法可以重新指定缓存有效期，例如：
$Cache->set('name', 'ThinkPHP', 3600); // 缓存name数据3600秒

```

[上一页](#)[下一页](#)

12.3 缓存队列

缓存队列

[上一页](#)[下一页](#)

新版的缓存支持缓存队列功能，有时候我们可能不需要那么多缓存数据，而只是需要保留最近的一些缓存数据，或者因为缓存容量问题，我们需要限制缓存的队列数据长度，这就可以使用缓存队列功能来解决。使用缓存队列很简单，只需要给当前缓存实例设置length参数即可，默认length参数为0，表示不启用缓存队列功能。下面的缓存队列的设置：

```
$Cache = Cache::getInstance('Xcache',array('expire'=>'60','length'=>10)); 或  
$Cache = Cache::getInstance('Xcache',array('expire'=>'60'));  
$Cache->setOptions('length',10); // 设置缓存队列长度为10  
者 $Cache->setOptions('queue','xcache'); // 设置缓存队列方式为xcache
```

[上一页](#)[下一页](#)

12.4 快捷缓存

快捷缓存

[上一页](#) [下一页](#)

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。（如果是3.1版本以上，建议用新增的cache方法替代

```
// 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方法)例如：`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');` S方法还支持对当前的缓存方式传入缓存参数，例如：

```
S('data',$Data,3600,'File',array('length'=>10,'temp'=>RUNTIME_PATH.'temp/'));
```

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次子目录来缓存。为了更加方便的操作缓存，3.1版本新增了cache函数用以设置和操作缓存。

使用方法：

1 缓存初始化 `cache(array('type'=>'xcache','expire'=>60));` 2 缓存设置

`cache('a',$value);` 3 缓存读取 `$value = cache('a');` 4 缓存删除 `cache('a',null);` 需要使用不同的缓存方式的时候 需要重新初始化，如果不初始化直接调用的话，则会按照系统配置自动初始化。

初始化的返回值，可以直接操作缓存：

```
$cache = cache(array('type'=>'xcache','expire'=>60));
$cache->set('name',$value);
$cache->get('name');
$cache->rm('name');
$cache = cache(array('type'=>'xcache','expire'=>60));
$cache->name = $value;
echo $cache->name;
unset($cache->name);
```

或者

[上一页](#) [下一页](#)

12.5 快速缓存

快速缓存

[上一页](#)[下一页](#)

S方法支持缓存有效期和缓存队列，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。

[上一页](#)[下一页](#)

12.6 查询缓存

查询缓存

[上一页](#)[下一页](#)

对于及时性要求不高的数据查询，我们可以使用查询缓存功能来提高性能，而且无需自己使用缓存方法进行缓存和获取。

新版内置的查询缓存功能支持所有的数据库，并且支持所有的缓存方式和有效期。

在使用查询缓存的时候，只需要调用Model类的cache方法，例如：

`$Model->cache(true)->select();` 如果使用了`cache(true)`，则在查询的同时会根据当前的查询SQL生成查询缓存，默认情况下缓存方式采用`DATA_CACHE_TYPE`参数设置的缓存方式（系统默认值为File表示采用文件方式缓存），缓存有效期是`DATA_CACHE_TIME`参数设置的时间，也可以单独制定查询缓存的缓存方式和有效期：`$Model->cache(true,60,'xcache')->select();` 表示当前查询缓存的缓存方式为xcache，并且缓存有效期为60秒。

同样的查询，如果没有使用cache方法，则不会获取或者生成任何缓存，即便是之前调用过Cache方法。

查询缓存只是供内部调用，如果希望查询缓存开放给其他程序调用，可以指定查询缓存的Key，例如：

`$Model->cache('cache_name',60)->select();` 则可以在外部通过S方法直接获取查询缓存的内容，`$value = S('cache_name');` 除了select方法之外，查询缓存还支持find和getField方法，以及他们的衍生方法（包括统计查询和动态查询方法）。具体应用的时候可以根据需要选择缓存方式和缓存有效期。

[上一页](#)[下一页](#)

12.7 SQL解析缓存

SQL解析缓存

[上一页](#)[下一页](#)

除了查询缓存之外，ThinkPHP还支持SQL解析缓存，因为ThinkPHP的ORM机制，所有的SQL都是动态生成的，然后由数据库驱动执行。

所以如果你的应用有大量的SQL查询需求，那么可以开启SQL解析缓存以减少SQL解析提高性能。要开启SQL解析缓存，只需要设置：`'DB_SQL_BUILD_CACHE' => true`，即可开启数据库查询的SQL创建缓存，默认缓存方式为文件方式，还可以支持xcache和apc方式缓存，只需要设置：

`'DB_SQL_BUILD_QUEUE' => 'xcache'`，我们知道，一个项目的查询SQL的量可能会非常巨大，所以有必要设置下缓存的队列长度，例如，我们希望SQL解析缓存不超过20条记录，可以设置：

`'DB_SQL_BUILD_LENGTH' => 20`，// SQL缓存的队列长度 注意：只有查询方法才支持SQL解析缓存

[上一页](#)[下一页](#)

12.8 静态缓存

静态缓存

[上一页](#) [下一页](#)

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，并且使用HTML_CACHE_RULES配置参数设置静态缓存规则文件。

静态规则的定义方式如下：

```
'HTML_CACHE_ON'=>true,
'HTML_CACHE_RULES'=> array(
    'ActionName'          => array('静态规则', '静态缓存有效期', '附加规则'),
    'ModuleName(小写)'    => array('静态规则', '静态缓存有效期', '附加规则')
    'ModuleName(小写):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 静态

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}',0)
```

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}',0) // 定义空模块的静态规则
```

```
'User:_empty'=>array('User/{:action}',0) // 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
```

静态规则是用于定义要生成的静态文件的名称，写法可以包括以下情况

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

格式：`{\$×××|function}` 例如：`{\$_GET.name}{\$_SERVER.REQUEST_URI|md5}` 2、使用框架特定的变量

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

本文档使用 [看云](#) 构建


```
{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，如果定义为0则表示永久缓存。

附加规则通常用于对静态规则进行函数运算，例如

```
'read'=>array('Think{id},{name}','60','md5') 翻译后的静态规则是  
md5('Think'.'$_GET[id]','.'$_GET[name']);
```

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 .html

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

[上一页](#)[下一页](#)

13. 扩展

扩展

[上一页](#)[下一页](#)

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。ThinkPHP的扩展目录位于框架的Extend目录下面，大部分扩展都放置到该目录下面，也有部分应用扩展位于项目类库目录下面。

下面是系统的扩展目录Extend下面的结构描述：

Action	控制器扩展	支持自动加载
Behavior	行为扩展	支持自动加载
Driver	驱动扩展，包括： Driver/Cache 缓存驱动 Driver/Db数据库驱动 Driver/Session SESSION驱动 Driver/TagLib标签库驱动 Driver/Template模板引擎驱动	支持自动加载
Engine	引擎扩展	入口定义后自动加载
Function	函数扩展	需要使用load手动加载
Library	类库扩展（包括ORG类库包和Com类库包）	可以配置自动加载
Mode	模式扩展	入口定义后自动加载
Model	模型扩展	支持自动加载
Tool	其他扩展或工具	不支持自动加载
Vendor	第三方类库扩展目录	可配置自动加载

后面我们会陆续介绍这些不同的扩展的使用方法，让你可以在不修改系统核心的情况下对框架和应用进行轻松的扩展。

[上一页](#)[下一页](#)

13.1 行为扩展

行为扩展

[上一页](#)[下一页](#)

行为和标签

行为在新版ThinkPHP的架构里面起着举足轻重的作用，在系统核心之上，设置了很多标签扩展位，而每个标签位置可以依次执行各自的独立行为。行为扩展就因此而诞生了，而且很多系统功能也是通过内置的行为扩展完成的，所有行为扩展都是可替换和增加的，由此形成了底层框架可组装的基础。

系统核心提供的标签位置包括下面几个（按照执行顺序排列）：

```
| app_init | 应用初始化标签位 |
|-----|-----|
| path_info | PATH_INFO检测标签位 |
| route_check | 路由检测标签位 |
| app_begin | 应用开始标签位 |
| action_name | 操作方法名标签位 |
| action_begin | 控制器开始标签位 |
| view_begin | 视图输出开始标签位 |
| view_template | 视图模板解析标签位 |
| view_parse | 视图解析标签位 |
| view_filter | 视图输出过滤标签位 |
| view_end | 视图输出结束标签位 |
| action_end | 控制器结束标签位 |
| app_end | 应用结束标签位 |
```

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

除了这些系统内置标签之外，开发人员还可以在应用中添加自己的应用标签，例如我们给应用的公共Action类CommonAction添加一个action_init标签位。

```
Class CommonAction extends Action{
    Public function _initialize(){
        tag('action_init'); // 添加action_init 标签
    }
}
```

注意：tag函数用于设置某个标签

位，可以传入并且只接受一个参数，如果需要传入多个参数，请使用数组，该参数为引用传值，所以只能传入变量。

内置行为

新版系统的很多核心功能也是采用行为扩展组装的，虽然在开发过程中可能感觉不到这种变化，但正是由本文档使用 [看云](#) 构建

于这种架构设计的改变，让新版变得更加灵活和易扩展，这是一个里程碑式的改变，对于满足项目日益纷繁复杂的需求和定制底层框架提供了更多的方便和可能性。

内置的行为包括下面：

行为名称	说明	对应标签位置
----- ----- -----		
checkRoute	路由检测行为，完成内置的路由功能	route_check
LocationTemplate	模板定位行为，完成模板文件自动定位和输出规则	view_template
ParseTemplate	模板文件解析，并支持第三方模板引擎驱动	view_parse
ShowPageTrace	页面Trace功能行为，完成页面Trace功能	view_end
ShowRuntime	运行时间显示行为，完成运行时间显示	view_filter
TokenBuild	令牌生成行为，完成表单令牌的自动生成	view_filter
ReadHtmlCache	读取静态缓存行为	app_init
WriteHtmlCache	生成静态缓存行为	view_filter

行为扩展

行为扩展首先是定义行为类，然后加入某个标签位置即可，内置的行为扩展就是一个很好的扩展示例。行为扩展类继承内置的行为基础类Behavior即可，类的命名规范是：

行为名称+Behavior

行为的执行入口方法是run方法，因此行为类只需要run方法具备public访问权限，入口方法只支持一个参数（可以用数组），并且采用引用方式传参，因此不需要任何返回值。每个行为可以定义options属性，该属性中的参数会自动转换成单独配置参数，下面是一个示例：

```
class TestBehavior extends Behavior {
    // 行为参数定义
    protected $options = array(
        'TEST_PARAM' => false,    // 行为参数 会转换成TEST_PARAM配置参数
    );
    // 行为扩展的执行入口必须是run
    public function run(&$params){
        if(C('TEST_PARAM')) {
            echo 'RUNTEST BEHAVIOR '.$params;
        }
    }
}
```

我们把TestBehavior行为扩展类放到项目目录的Lib/Behavior目录下面，其中options属性必须是数组方式定义，而且在行为初始化的同时options属性中的参数会转换成全局的配置参数，所以，我们在入口方法里面可以直接使用：C('TEST_PARAM') TEST_PARAM参数是没有预先定义的，行为的options属性里面提供的参数只是一个初始值，所有的参数我们可以在项目配置文件中重新定义。例如，上面的Test行为如果要执行，我们需要在项目配置文件中添加下面的配置： 'TEST_PARAM' =>true

行为调用

定义了一个行为扩展之后，系统提供了三种方式可以调用行为：

一、添加到行为配置文件

为了执行Test行为，我们在这里把Test行为加入app_end 标签位置，在项目配置目录中添加tags.php文

件，在其中添加下面代码：

```
return array(
    'app_end'=>array('Test'), // 在app_end 标签位添加Test行为
);
```

tags文件是项目的行

为定义文件，在这个文件中可以给每个标签位置添加项目自己的扩展行为，当然你也可以添加到应用自己的标签位，例如，我们可以把Test行为添加到前面项目自己定义的action_init标签位置：

```
return array(
    'action_init'=>array('Test'), // 在action_init 标签位添加Test行为
);
```

应用行为

的定义没有限制，你甚至可以把同一个行为放到多个标签位置执行多次，例如：

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

默认情况下应用

行为扩展会并入系统行为扩展一起执行，也就是说如果系统的行为定义中app_end标签中已经定义了其他行为，会首先执行系统行为扩展，然后再执行应用行为扩展，如果你希望项目的行为扩展完全替换系统的

```
return array(
    'app_end'=>array('Test', '_overlay'=>1),
```

行为扩展，可以使用：);

表示app_end标签

位用定义的应用行为完全替换该位置的系统行为，也就是说即使系统标签在app_end定义了其他的系统行为也不会被执行，但是必须注意，行为的命名不要和系统行为一样。

二、函数方式动态定义

除了定义tags行为配置文件之外，系统还提供了动态添加行为到标签位的方法，例如我们可以使用下面的方式添加Test行为到app_end标签位，而无需在tags文件中添加定义：

```
add_tag_behavior('app_end','Test');
```

表示把Test行为添加到app_end标签位的最后，你可以把这个代码放到项目的公共函数文件中甚至直接放到行为类的最后（如果你确定这个行为扩展只有你的项目会用到的话）。

应用行为扩展类除了放到项目类库的Behavior目录外，还可以放到系统的行为扩展目录Extend/Behavior/，行为类在执行过程中会自动加载。

三、不放入标签直接执行

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

```

class AuthCheckBehavior extends Behavior {
  // 行为参数定义
  protected $options = array(
    'USER_AUTH_ON'      => false,    // 是否开启用户认证
    'USER_AUTH_ID'      => 'user_id', // 定义用户的id为权限认证字段
  );
  // 行为扩展的执行入口必须是run
  public function run(&$return){
    if(C('USER_AUTH_ON ')) {
      // 进行权限认证逻辑 如果认证通过 $return = true;
      // 否则用halt输出错误信息
    }
  }
}

```

定义了

AuthCheck行为后，然后在_initialize方法中直接用下面的方式调用：`B('AuthCheck');`

注意：因为这种方式的行为调用需要在相关位置添加代码，所以一般只有在应用代码才直接使用B方法调用。

[上一页](#)[下一页](#)

13.2 类库扩展

类库扩展

[上一页](#)[下一页](#)

类库扩展包括基类库扩展、应用类库扩展和第三方类库扩展，所有扩展类库不会自动加载，需要手动加载或者定义别名和配置自动加载（详细可以参考4.2.3类库导入和4.2.5自动加载）。

基类库扩展

目前支持的基类库扩展包括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，你甚至还可以创建属于自己企业的类库，只需要在Extend/Library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Extend/Library/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');`；目前官方提供的扩展或者第三方扩展都在ORG类库包下面。

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，也只有在项目类库目录下面增加的类库才能使用import方法导入。例如，我们在MyApp的项目类库目录Lib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');
```

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，我们建议你放到第三方类库扩展目录Extend/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');`；需要注意的是，vendor方法默认导入的类库后缀是php的而不是class.php的，如果你的第三方类库的后缀是class.php，可以使用：`Vendor('Zend.Filter.Dir', '', '.class.php');`；或者使用：

`import('Zend.Filter.Dir', VENDOR_PATH);`；通过使用第三方类库扩展，我们可以直接使用Zend、CI或者其他框架中的类库。

[上一页](#)[下一页](#)

13.3 控制器扩展

控制器扩展

[上一页](#)[下一页](#)

系统内置的Action基础类完成的功能有限，有时候，我们在项目经常需要扩展一个用于项目的公共Action，又或者我们需要为某些特殊应用增加功能，这些都可以使用控制器扩展来实现。

控制器扩展接口

系统Action类提供了一个初始化方法_initialize接口，可以用于扩展需要，_initialize方法会在所有操作方法调用之前首先执行，用法：

_initialize 控制器初始化方法	
用法	_initialize()
参数	无
返回值	无
相关方法	可以和getActionName方法配合使用

除了初始化接口外，Action类还提供了两个用于行为扩展的标签位置action_begin和action_end，因此你还可以通过行为扩展来扩展控制器的功能。

控制器扩展只需要继承Action，例如：``Class ExtendAction extends Action{`

```
Public function _initialize(){
    // 初始化的时候检查用户权限
    $this->checkRbac();
}

// 检查用户权限
protected function checkRbac() {
    // 这里是具体的检测代码
}

// 添加新的上传操作方法
protected function upload() {
    // 这里是具体的上传实现代码
}
```

`}``在有些情况下面，控制器扩展并不一定要继承基础的Action。

Hack方法

新版提供了两个hack方法用于对模块和操作方法进行扩展，这些hack函数可以定义到项目的公共函数库里

面。

hack_module 模块hack函数	
用法	__hack_module ()
参数	无
返回值	如果返回一个对象，则会继续执行该对象的对应当前操作的方法。否则，将在执行完hack_module函数后中止当前操作的执行，但不影响app_end标签的行为执行。

hack_module仅在访问一个不存在的模块的时候会被调用，优先级大于空模块。简单的说，如果定义了__hack_module 则当前模块不存在的情况下操作会被接管。

下面是一个定义的示例：`function hack_module(){

```
    if ('Test'== MODULE_NAME){
        $module = New MyAction();
        return $module;
    }
}
```

<table border="0" cellspacing="1" cellpadding="0"><tr><th colspan="2">__hack_
__hack_action函数定义后仅在访问一个不存在的操作方法，而且当前控制器没有定义空操作方法和对
下面是一个定义的示例：

```
function __hack_action(){
    if ('Test'== ACTION_NAME){
        echo 'Hello,Just Test! You can do anything here...';
    }
}
```

[上一页](#)[下一页](#)

13.4 模型扩展

模型扩展

[上一页](#) [下一页](#)

模型扩展目录位于Extend/Model下面，ThinkPHP本身提供了丰富的模型扩展，例如：

模型名	名称	说明
AdvModel	高级模型	扩展了文本字段、只读字段、序列化字段、延迟写入、乐观锁等高级特性
ViewModel	视图模型	扩展了模型的视图操作功能
RelationModel	关联模型	扩展了模型的关联操作
MongoModel	Mongo模型	扩展了对Mongo数据库的数据操作支持

这些扩展模型都是基于系统的基础模型类Model扩展而来。

模型扩展接口

ThinkPHP的新版基础模型类Model具有很好的扩展性，对模型的CURD方法都提供了扩展接口，包含：

接口名称	所属方法	接口方法（参数）
初始化接口	全局	_initialize()
表达式过滤接口	全局	_options_filter(&\$options)
写入前置接口	add方法	_before_insert(&\$data,\$options)
写入后置接口	add方法	_after_insert(\$data,\$options)
更新前置接口	save方法	_before_update(&\$data,\$options)
更新后置接口	save方法	_after_update(\$data,\$options)
数据写入接口	add、save方法	_facade(\$data)
数据库切换接口	db方法	_after_db()
删除后置接口	delete方法	_after_delete(\$data,\$options)
查询后置接口	select方法	_after_select(&\$result,\$options)
查询后置接口	find方法	_after_find(&\$result,\$options)

目前提供的扩展模型包括：高级模型（AdvModel）、视图模型（ViewModel）、关联模型（RelationModel）和Mongo模型都是继承Model类并且都通过了扩展完成了很多其他的功能。不过在某些情况下，模型扩展并不一定要继承基础模型Model。

调用扩展模型

定义了模型扩展之后，有多种方式可以使用扩展模型：

一、继承扩展模型

最普遍的用法就是项目中的自定义模型或者公共模型直接继承扩展模型，例如：

我们需要使用Mongo模型的话，可以：`Class UserModel extends MongoModel{}` 把原来的继承

从Model类改为扩展模型MongoModel，就可以使用MongoModel的所有功能。

如果你的项目大部分模型都继承了一个公共的模型类CommonModel的话，只需要改下CommonModel的继承定义：`Class CommonModel extends MongoModel{}` 所有继承自CommonModel的自定义模型也可以使用MongoModel的功能。

二、使用动态模型切换的方式

例如，我们定义了一个UserModel如下：`Class UserModel extends Model{}` 为了使用AdvModel高级模型的功能，我们使用下面的方式切换到高级模型进行操作：

`$User->switchModel("Adv")->top10();` 注意：动态模型切换方法switchModel调用的时候无需写完整的扩展模型名称，需要去掉扩展模型的Model后缀后调用。

三、M方法实例化

如果我们没有定义自定义模型，则可以直接采用M方法实例化需要继承的扩展模型，例如：

`M("AdvModel:User")->top10();`

这里表示实例化User模型，而且该模型使用的基础模型类为AdvModel扩展模型类，这里引用的扩展模型需要使用全名。

[上一页](#) [下一页](#)

13.5 驱动扩展

驱动扩展

[上一页](#) [下一页](#)

这里说的驱动扩展是一种泛指，驱动扩展的目录位于扩展目录Extend/Driver，包括数据库驱动、缓存驱动、标签库驱动和模板引擎驱动。

数据库驱动

数据库抽象层的设计是由抽象数据库类（Db）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、Mongo以及PDO驱动类，可以满足常用的数据库操作的需要。

数据库驱动扩展目录位于系统扩展目录Extend/Driver/Db，如果需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，并放到系统扩展目

```
Class DbOdbc extends Db{
}
```

录 Extend/Driver/Db目录下。 每个数据库驱动必须要实现的方法包括（具体参数可以参考现有的数据库驱动类库）：

| 驱动方法 | 方法说明 |

|-----|-----|

| 架构方法 | __construct(\$config="") |

| 数据库连接方法 | connect(\$config="",\$linkNum=0,\$force=false) |

| 释放查询方法 | free() |

| 查询操作方法 | query(\$str) |

| 执行操作方法 | execute(\$str) |

| 开启事务方法 | startTrans() |

| 事务提交方法 | commit() |

| 事务回滚方法 | rollback() |

| 获取查询数据方法 | getAll() |

| 获取字段信息方法 | getFields(\$tableName) |

| 获取数据库的表 | getTables(\$dbName="") |

| 关闭数据库方法 | close() |

| 获取错误信息方法 | error() |

| SQL安全过滤方法 | escapeString(\$str) |

数据库的CURD接口方法（通常这些方法无需重新定义）

| 方法 | 说明 |

|-----|-----|

写入	insert(\$data,\$options=array(),\$replace=false)
更新	update(\$data,\$options)
删除	delete(\$options=array())
查询	select(\$options=array())

介于不同数据库的查询方法存在区别，所以经常需要对查询的语句进行重新定义，这就需要修改针对查询的selectSql属性。该属性定义了当前数据库驱动的查询表达式，默认的定义是：

```
'SELECT%DISTINCT% %FIELD% FROM  
%TABLE%%JOIN%%WHERE%%GROUP%%HAVING%%ORDER%%LIMIT% %UNION%'
```

驱动可以更改或者删除个别查询定义，或者更改某个替换字符串的解析方法，这些方法包括：

方法名	说明	对应
parseTable	数据库表名解析	%TABLE%
parseWhere	数据库查询条件解析	%WHERE%
parseLimit	数据库查询Limit解析	%LIMIT%
parseJoin	数据库JOIN查询解析	%JOIN%
parseOrder	数据库查询排序解析	%ORDER%
parseGroup	数据库group查询解析	%GROUP%
parseHaving	数据库having解析	%HAVING%
parseDistinct	数据库distinct解析	%DISTINCT%
parseUnion	数据库union解析	%UNION%
parseField	数据库字段解析	%FIELD%

驱动的其他方法根据自身驱动需要和特性进行添加，例如，有些数据库的特殊性，需要覆盖父类Db类中的解析和过滤方法，包括：

parseKey	数据库字段名解析
parseValue	数据库字段值解析
parseSet	数据库set分析
parseLock	数据库锁机制

定义了驱动扩展后，需要使用的时候，设置相应的数据库类型即可：

```
'DB_TYPE'=>'odbc', // 数据库类型配置不区分大小写
```

缓存驱动

系统的缓存实现是由缓存类和缓存驱动组成，缓存驱动扩展位于Extend/Driver/Cache目录下面，目前已经提供了包括APC、Db、Memcache、Shmop、Sqlite、Redis、Eaccelerator和Xcache缓存方式的驱动扩展，缓存驱动必须继承Cache类，缓存驱动类的命名规范是：

Cache+驱动类名称（首字母大写）

并实现下面的驱动接口：

方法说明	接口方法
架构方法	__construct(\$options=')
读取缓存	get(\$name)
写入缓存	set(\$name,\$value,\$expire=null)
删除缓存	rm(\$name)
清空缓存	clear()

注意：有些缓存方式并未提供清空缓存接口，可以无需定义。

所有缓存驱动的有效期参数约定，如果设置为0 则表示永久缓存。如果要让缓存驱动支持缓存队列功能，需要在缓存接口的set操作方法设置成功后添加如下代码：

```
if($this->options['length']>0) {
    // 记录缓存队列
    $this->queue($name);
}
```

Session驱动

新版支持对session 的handler驱动，可以通过驱动更改session的管理机制。Session驱动扩展目录位于Extend/Driver/Session下面，命名规范是：

Session+驱动类名称（首字母大写）

并实现下面的驱动接口：

方法说明	接口方法
执行入口	execute()并且在方法中调用session_set_save_handler函数指定handler操作机制
并建议添加下面的接口方法	
打开Session	open(\$savePath,\$sessionName)
关闭Session	close()
读取Session	read(\$id)
写入Session	write(\$id,\$data)
删除Session	destory(\$id)
Session过期回收	gc(\$maxlifetime)

标签库驱动

任何一个模板引擎的功能都不可能是为你量身定制的，具有一个良好的可扩展机制也是模板引擎的另外一个考量，Smarty采用的是插件方法来实现扩展，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。

每个标签库对应一个标签库驱动类，每个驱动类负责对标签库中的所有标签的解析。标签库驱动类的作用
本文档使用 [看云](#) 构建

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）。

系统的标签库驱动扩展目录位于Extend/Driver/TagLib目录下面，命名规范是：

TagLib+标签库名称（首字母大写）

目前已经提供了Html标签库驱动支持，标签库驱动扩展必须继承TagLib类，例如我们扩展一个Test标签库：Class TagLibTest extends TagLib{} 首先需要定义标签库的标签定义，标签定义包含了所有标签库中支持的所有标签，定义方式如下：

```
protected $tags = array(
    // 定义标签
    'input'=>array('attr'=>'type,name,id,value','close'=>0), // input标签
);
```

标签库

的所有支持标签都在tags属性中进行定义，tags属性是一个二维数组，每个元素就是一个标签定义，索引名就是标签名，采用小写定义，调用的时候不区分大小写。

每个标签定义支持的属性包括：

| 属性名 | 说明 |

|-----|-----|

| attr | 标签支持的属性列表，用逗号分隔 |

| close | 标签是否为闭合方式（0闭合 1不闭合），默认为不闭合 |

| level | 标签的嵌套层次（只有不闭合的标签才有嵌套层次） |

| alias | 标签别名 |

定义了标签属性后，就需要定义每个标签的解析方法了，每个标签的解析方法在定义的时候需要添加“_”前缀，可以传入两个参数，属性字符串和内容字符串（针对非闭合标签）。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input解析方法的定义：

```
public function _input($attr,$content) {
    $tag = $this->parseXmlAttr($attr,'input');
    $name = $tag['name'];
    $id = $tag['id'];
    $type = $tag['type'];
    $value = $this->autoBuildVar($tag['value']);
    $str = '';
    return $str;
}
```

在每个标签的解析方法

中，首先需要调用 \$this->parseXmlAttr(\$attr,'input'); 表示分析input标签的标签定义，并返回input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。由于是示例，我们没有对标签中的全部变量进行解析，只是支持了value属性的变量传入。

定义好标签库扩展之后，我们就可以在模板中使用了，首先我们必须告诉模板申明Test标签库，用taglib标签，例如：<taglib name='Test' /> name属性支持申明多个标签库，用逗号分隔即可。申明Test标签库之后，就可以使用Test标签库中的所有标签库了，调用方式如下：

<test:input type='radio' id='test' name='mail' value='value' /> 注意：调用扩展标签库的标签的时候，必须加上标签库的XML命名空间前缀。

由于我们定义的input标签是闭合标签，如果是非闭合方式的话，应该是写成：

```
<test:input type='radio' id='test' name='mail' value='value'></test:input>
```

Input标签定义value属性可以支持变量传入，所以value被认为是一个变量名，如果在Action中已经给本文档使用 [看云](#) 构建

value模板变量赋值，例如：`$this->assign('value','my test value');`；最后标签被模板引擎编译后，就会输出：

```
<input type='radio' id='test' name='mail' value='my test value' />
```

模板引擎驱动

系统支持模板引擎的扩展机制，模板引擎驱动的扩展目录位于Extend/Driver/Template下面，并且命名规范为：

Template+模板引擎名（首字母大写）

模板引擎驱动不需要继承任何类库，并且只需要实现一个接口方法：

接口说明	使用说明
渲染模板输出	fetch(\$templateFile,\$var) templateFile表示要解析的模板文件名 var表示要传入的模板变量数组

官方目前已经提供了包括Smarty、EaseTemplate、TemplateLite和Smart在内的第三方模板引擎扩展。模板引擎扩展必须要配合第三方类库一起使用，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，下载最新的Smarty模板引擎文件放到系统目录的Vendor第三方类库目录下面，建立Smarty子目录。

然后，修改项目配置文件，把模板引擎改为扩展的模板引擎名：`'TMPL_ENGINE_TYPE' =>'Smarty'`就可以用smarty标签来定义你的模板文件了，改变模板引擎驱动并不会影响系统内部的模板变量输出和模板文件定位，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,
)
```

一般情况下，无需设置TMPL_ENGINE_CONFIG参数，模板引擎驱动已经有最适合的默认值了。

[上一页](#)[下一页](#)

13.6 Widget扩展

Widget扩展

[上一页](#) [下一页](#)

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('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。

如果使用了renderFile方法调用了模板，那么在模板中就可以使用：

`{count}` 来输出w方法传入的变量。

如果w方法传入的数据是 `array('id'=>5, 'name'=>'thinkphp')`；那么widget模板中就可以输出id和name两个变量。

可以理解成W方法传入的参数是

`array('模板变量1'=>值1, '模板变量2'=>'值2'...)`

注意：模板中的变量由renderFile方法的var变量决定，并非取决于W方法传入的参数，render方法本身可以对W方法传入的参数进行处理后传给renderFile方法，尽管大多数情况下都是直接传入data变量到renderFile方法中去。

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。

Widget的模板文件单独存放，放置到当前项目的Lib/Widget/ShowComment/目录下面，取决于rendFile方法如何调用，默认情况下，是调用和widget同名的模板文件，例如当前Widget是

ShowCommentWidget , 其中代码如下 : `$this->renderFile()` ; 则调用的widget模板位于 Lib/Widget/ShowComment/ShowComment.html ,
如果调用 `$this->renderFile('comment')` ; 调用的widget模板则位于 Lib/Widget/ShowComment/comment.html ,
如果需要调用子目录下面的模板 , 则采用 `$this->renderFile('article/comment')` ; 调用的 widget模板则位于 Lib/Widget/ShowComment/article/comment.html。

[上一页](#)[下一页](#)

13.7 模式扩展

模式扩展

[上一页](#)[下一页](#)

模式扩展属于系统核心级别的扩展，可以改变底层的架构体系。在众多扩展中，也只有模式扩展具有改变和替换核心MVC的可能，其他扩展只是在标准模式基础之上的增强和替换，无法从根本上改变底层的架构。所以，大家会看到不同的模式扩展可能具有很大的用法区别，有些模式扩展是为某个特别的应用环境而定制的，例如CLI模式、AMF模式和PHPRPC模式。

新版对模式扩展的改进和行为扩展的增强使得开发人员对底层框架的DIY更加方便，也正是因为支持对框架底层进行DIY，使得ThinkPHP能够满足企业开发中更加复杂的项目需求。

使用模式扩展

我们前面所涉及的所有用法都是基于框架内置的标准模式的，除了标准模式之外，官方还提供了一些常用的模式扩展，模式扩展的目录位于Extend/Mode下面，已经提供的包括：Cli（命令模式）、Lite（精简模式）、Thin（简洁模式）、AMF模式、PHPRPC模式和REST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。

要使用某个扩展模式，需要修改项目的入口文件，添加一行定义代码：

```
define('MODE_NAME', '模式扩展名称');
```

每个项目只能使用一个模式扩展，所以即使采用了分组，不同的分组也只能采用相同的模式扩展。使用了模式扩展后，项目的编译缓存文件有所变化，例如，如果你当前用的是REST模式，那么生成的编译缓存文件则会变成~rest_runtime.php。

具体不同模式的用法需要参考每个模式扩展的帮助文件。

简洁模式

简洁模式相当于标准模式的主要区别在于：

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('MODE_NAME', 'Thin'); //采用简洁模式运行
```

精简模式

精简模式在简洁模式的基础上，增加了：

默认使用PHP模板；

本文档使用 [看云](#) 构建

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('MODE_NAME', 'Lite');//采用精简模式运行
```

命令模式

命令模式用于支持命令行模式下面的PHP应用，需要在入口文件设置：

```
define('MODE_NAME', 'cli') //采用CLI运行模式运行
```

在命令模式下面，支持两种命令行的参数模式，

一、PATHINFO参数模式（URL_MODEL为1）

在PATHINFO参数模式下面，我们可以这样调用模块和操作 `index.php module/action/id/4` 二、普通参数模式（URL_MODEL设置为其它）

在普通参数模式下面，我们需要这样调用模块和操作 `index.php module action id 4` 在命令行模式下面，系统会自动把参数转换为GET变量，无论采用哪种命令行参数模式，我们可以直接使用GET变量获取参数，例如，采用下面的方式调用 `index.php Info/read/category/2/id/4` 在控制器中，我们可以直接获取`$_GET['category']`（这里传入的是2）和`$_GET['id']`（这里传入的是4）参数，如果你需要自己解析传入的参数顺序和值，就需要采用原生的系统变量`$_SERVER['argv']`来获取参数了。

AMF模式

AMF模式采用了ZendAMF类库，支持AMF开发和Flash进行通讯。首先，我们需要设置当前运行模式为Amf模式，在入口文件中增加下面代码：`define('MODE_NAME', 'amf')` //采用Amf运行模式运行然后在项目配置文件中定义

```
'APP_AMF_ACTIONS'=>'Index,User,Shop...'// 定义AMF模式的模块列表 只有在APP_AMF_ACTIONS中定义的模块才能在Amf模式中调用到。
```

最后一步就是在你的Flash客户端或者AS脚本中修改Amf的网关地址为当前项目的入口地址即可。

PHPRPC模式

首先，我们设置当前运行模式为Phprpc模式：

```
define('MODE_NAME', 'phprpc') //采用Phprpc运行模式运行
```

然后在项目配置文件中定义

```
'APP_PHPRPC_ACTIONS'=>'Index,User,Shop...'// 定义PHPRPC模式的模块列表 只有在APP_PHPRPC_ACTIONS中定义的模块才能在PHPRPC模式中调用到。
```

REST模式

Rest模式主要是为了支持RESTFul的开发，鉴于目前Rest主要用来提供接口服务，所以单独作为模式扩展来使用。首先，我们设置当前运行模式为rest模式：

```
define('MODE_NAME', 'rest') //采用rest模式运行
```

关于Rest的更多用法，请参考16.2 REST支持部分。

定制模式扩展

要定制自己的模式扩展，首先要定义模式扩展的定义文件，定义文件位于Extend/Mode目录下，命名就是模式扩展的名称（全部为小写），定义文件是一个数组，包括：

| core | 系统核心列表文件定义 |

|-----|-----|

| config | 模式配置文件 |

| alias | 模式别名定义文件 |

| extends | 模式系统行为定义 |

| tags | 应用行为定义文件 |

core是模式核心列表文件，如果core没有定义，则表示采用标准模式的核心列表文件，被定义的文件列表会纳入编译缓存，核心列表可以包含函数文件和类库文件，注意下面的文件无需定义：系统的Common公共文件、Think类、ThinkException类和Behavior类。下面的类必须定义：App类和Action类。

config是模式配置定义，可以采用文件名或者直接用数组定义的方式。

alias是模式别名定义，可以采用文件名或者直接用数组定义的方式。

extends是模式系统行为定义，可以采用文件名或者直接用数组定义的方式。

tags是应用行为定义，可以采用文件名或者直接用数组定义的方式。

上面这些定义，只有需要的时候才要定义，如果没有则可不定义，一般core定义是模式扩展必须的，改变核心列表文件的定义就能起到自定义MVC的目的。

例如，命令行模式的模式定义文件为：

```
// 命令行模式定义文件
return array(
    'core' => array(
        MODE_PATH.'Cli/functions.php',    // 命令行系统函数库
        MODE_PATH.'Cli/Log.class.php',
        MODE_PATH.'Cli/App.class.php',
        MODE_PATH.'Cli/Action.class.php',
    ),
    // 项目别名定义文件 [支持数组直接定义或者文件名定义]
    'alias' => array(
        'Model' => MODE_PATH.'Cli/Model.class.php',
        'Db' => MODE_PATH.'Cli/Db.class.php',
        'Cache' => CORE_PATH.'Core/Cache.class.php',
        'Debug' => CORE_PATH.'Util/Debug.class.php',
    ),
    // 系统行为定义文件
    'extends' => array(),
);
```

模式扩展本身是一个扩展的集成，自身还可以包含其他扩展，例如行为扩展、函数扩展、类库扩展等。如果模式扩展中包含了自己的行为扩展，那么可以放到模式扩展目录下面的Behavior目录下面，系统可以自动加载该目录下面的行为类库。

[上一页](#)[下一页](#)

13.8 引擎扩展

引擎扩展

[上一页](#)[下一页](#)

引擎扩展是比模式扩展更高层次的扩展机制，顾名思义，引擎扩展是一个框架的引擎替换，当模式扩展无法满足运行环境的特殊要求的时候，就需要使用引擎扩展。引擎扩展的一个最大特点就是替换了原有的框架入口文件，是为了满足日益发展的App Engine平台而诞生的。

新版的第一个引擎扩展就是SAE扩展，由于新浪SAE环境的特殊要求，内置的标准模式中的文件写入操作无法在SAE环境中使用，因此导致系统的框架入口ThinkPHP.php中的编译缓存无法实现，必须通过引擎扩展替换框架入口才可以实现。

关于SAE的支持和使用，请参考17 SAE支持部分。

[上一页](#)[下一页](#)

14. 安全

安全

[上一页](#)[下一页](#)

在项目开发完成准备部署之前，应该检查下是否存在安全隐患，这一部分内容帮助你一起来加强项目的安全问题，指导你如何使用表单令牌、字段类型验证、输入过滤、上传安全、防止XSS攻击和目录安全保护等功能。

[上一页](#)[下一页](#)

14.1 表单令牌

表单令牌

[上一页](#)[下一页](#)

ThinkPHP内置了表单令牌验证功能，可以有效防止表单的重复提交等安全防护。

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
'TOKEN_RESET'=>true,    //令牌验证出错后是否重置令牌 默认为true 如果开启表单令牌验证功
```

能，系统会自动在带有表单的模板文件里面自动生成以TOKEN_NAME为名称的隐藏域，其值则是TOKEN_TYPE方式生成的哈希字符串，用于实现表单的自动令牌验证。

自动生成的隐藏域位于表单Form结束标志之前，如果希望自己控制隐藏域的位置，可以手动在表单页面添加{TOKEN} 标识，系统会在输出模板的时候自动替换。

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

如果个别页面输出不希望进行表单令牌验证，可以在控制器中的输出方法之前动态关闭表单令牌验证，例

```
C('TOKEN_ON', false);
```

如：`$this->display();` 模型类在创建数据对象的同时会自动进行表单令牌验证操作，如果你没有使用create方法创建数据对象的话，则需要手动调用模型的autoCheckToken方法进行表单令牌验证。如果返回false，则表示表单令牌验证错误。例如：

```
$User = M("User"); // 实例化User对象
// 手动进行令牌验证
if (!$User->autoCheckToken($_POST)){
// 令牌验证错误
}
```

[上一页](#)[下一页](#)

14.2 字段类型验证

字段类型验证

[上一页](#) [下一页](#)

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

```
$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
```

例如：\$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

```
$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
```

例如：\$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。

[上一页](#) [下一页](#)

14.3 防止SQL注入

防止SQL注入

[上一页](#) [下一页](#)

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` 即使用户输入了一些恶意的id参数，系统也会强制转换成整型，避免恶意注入。这是因为，系统会对数据进行强制的数据类型检测，并且对数据来源进行数据格式转换。而且，对于字符串类型的数据，ThinkPHP都会进行`escape_string`处理(`real_escape_string`,`mysql_escape_string`)。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于客户端的用户输入，要有效的防止SQL注入问题，我们建议：

- 查询条件尽量使用数组方式，这是更为安全的方式；
- 如果不得已必须使用字符串查询条件，使用预处理机制（3.1版本新增特性）；
- 开启数据字段类型验证，可以对数值数据类型做强制转换；（3.1版本开始已经强制进行字段类型验证了）
- 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相关部分已经有详细的描述。

查询条件预处理

`where`方法使用字符串条件的时候，支持预处理（安全过滤），并支持两种方式传入预处理参数，例如：
`$Model->where("id=%d and username='%s' and xx='%f'",array($id,$username,$xx))`
 或者
`$Model->where("id=%d and username='%s' and xx='%f'", $id,$username,$xx)->select`
 模型的`query`和`execute`方法 同样支持预处理机制，例如：
`$model->query('select * from user where id=%d and status=%d',$id,$status);` 或者
`$model->query('select * from user where id=%d and status=%d',array($id,$status)`
`execute`方法用法同`query`方法。

[上一页](#) [下一页](#)

14.4 输入过滤

输入过滤

[上一页](#) [下一页](#)

永远不要相信客户端提交的数据，所以对于输入数据的过滤势在必行，我们建议：

- 开启令牌验证避免数据的重复提交；
- 使用自动验证和自动完成机制进行初步过滤；
- 使用系统Action类提供的_get_post_cookie等方法获取数据；
- 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；

系统变量的全局过滤

系统变量的全局过滤功能，采用VAR_FILTERS 定义，默认为空，表示不进行任何过滤。

如果设置了VAR_FILTERS参数，对GET POST系统变量会进行过滤，例如：

`'VAR_FILTERS'=>'htmlspecialchars'` 也可以支持多个方法过滤，例如：

`'VAR_FILTERS'=>'stripslashes,strip_tags'` 注意如果系统变量存在多维数组的情况，设置的过滤方法要能够很好的支持多维数组过滤。

表单数据合法性检测

使用create方法创建数据对象的时候，可以使用数据的合法性检测，有两种方式：

一、可以配置insertFields 和 updateFields属性

可以分别为新增和编辑表单设置insertFields 和 updateFields属性

使用create方法创建数据对象的时候，不在定义范围内的属性将直接丢弃，避免表单提交非法数据。

insertFields 和 updateFields属性的设置采用字符串（逗号分割多个字段）或者数组的方式。

设置的字段应该是实际的数据表字段，而不受字段映射的影响。例如：

```
class UserModel extends Model{
    protected $insertFields = array('account','password','nickname','email');
    protected $updateFields = array('nickname','email');
}
```

定义后，调用add方法写入用户数据的时候，只能写入'account','password','nickname','email' 这几个字段，编辑的时候只能更新'nickname','email'两个字段。

在使用的时候，我们调用create方法的时候，会根据提交类型自动识别insertFields和updateFields属性：`D('User')->create()`；二、直接调用field方法

如果不想定义insertFields和updateFields属性，可以在调用create方法之前直接调用field方法，例如，实现和上面的例子同样的作用：

在新增用户数据的时候，使用：

`M('User')->field('account,password,nickname,email')->create();` 而在更新用户数据的时候, 使用: `M('User')->field('nickname,email')->create();` 这里的字段也是实际的数据表字段。

field方法也可以使用数组方式。

使用字段合法性检测后, 你不再需要担心用户在提交表单的时候注入非法字段数据了。

写入数据过滤

可以在数据写入数据库之前调用filter方法对数据进行安全过滤, 例如:

```
$this->data($data)->filter('strip_tags')->add();
```

[上一页](#)[下一页](#)

14.5 上传安全

上传安全

[上一页](#)[下一页](#)

网站的上传功能也是一个非常容易被攻击的入口，所以对上传功能的安全检查是尤其必要的。系统提供的上传扩展类库提供了安全方面的支持，包括对文件后缀、文件类型、文件大小以及上传图片文件的合法性检查，确保你已经在上传操作中启用了这些合法性检查。

[上一页](#)[下一页](#)

14.6 防止XSS攻击

防止XSS攻击

[上一页](#)[下一页](#)

XSS（跨站脚本攻击）可以用于窃取其他用户的Cookie信息，要避免此类问题，可以采用如下解决方案：

- 直接过滤所有的JavaScript脚本；
- 转义Html元字符，使用htmlentities、htmlspecialchars等函数；
- 系统的扩展函数库提供了XSS安全过滤的remove_xss方法；
- 新版对URL访问的一些系统变量已经做了XSS处理。

[上一页](#)[下一页](#)

14.7 其他安全建议

其他安全建议

[上一页](#)[下一页](#)

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）；

[上一页](#)[下一页](#)

14.8 目录安全文件

目录安全文件

[上一页](#)[下一页](#)

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。

为了添加目录安全文件，我们需要在入口文件里面定义了BUILD_DIR_SECURE 常量，例如：

`define('BUILD_DIR_SECURE', true);` 重新运行项目后会自动给项目的相关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!');
```

[上一页](#)[下一页](#)

14.9 保护模板文件

保护模板文件

[上一页](#)[下一页](#)

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。

```
<Files *.html>
Order Allow,Deny
Deny from all
</Files>
```

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。

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。

[上一页](#)[下一页](#)

15. 性能

性能

[上一页](#)[下一页](#)

在部署到生产环境之前，我们可以对系统的性能进行可能的调优。

[上一页](#)[下一页](#)

15.1 关闭调试模式

关闭调试模式

[上一页](#)[下一页](#)

首先确认你已经关闭了调试模式，由于关闭调试模式之后，系统会自动生成项目编译缓存以及关闭日志写入，这样可以减少很多的IO加载和日志写入的开销。

要关闭调试模式，只需要在入口文件中删除常量APP_DEBUG的定义或者定义为false。关闭调试模式即表示启用部署模式。

以官方的Hello 示例和blog示例首页为例进行测试调试模式开启和关闭的对比数据：

Hello示例	开启调试模式	关闭调试模式
加载文件	29	5
运行时间	0.0231s	0.0023s
内存占用	817kb	800kb
每秒请求次数	127.35	188.54

blog示例	开启调试模式	关闭调试模式
加载文件	41	19
运行时间	0.1584s	0.0514s
内存占用	2,189kb	2,162kb
每秒请求次数	28.17	52.25

注：为确保数据准确，每次测试都重启Apache服务。

开启页面压缩输出

3.1版本开始，增加了OUTPUT_ENCODE配置参数，用于控制页面压缩输出。

会自动检测zlib.output_compression配置，如果php.ini里面zlib.output_compression没有开启，并且OUTPUT_ENCODE配置开启 则会进行页面压缩输出。

[上一页](#)[下一页](#)

15.2 开启缓存

开启缓存

[上一页](#)[下一页](#)

给你的部署环境安装APC或者XCache缓存能够有效的提高运行性能和内存占用。
还是以官方的blog示例首页为例进行测试部署模式下面开启XCache缓存前后的对比数据：

Hello示例	Xcache关闭	Xcache开启
运行时间	0.0023s	0.0016s
内存占用	800 kb	104 kb
每秒请求次数	188.54	427.35

blog示例	Xcache关闭	Xcache开启
运行时间	0.0514s	0.0245s
内存占用	2,162 kb	418 kb
每秒请求次数	52.25	98.65

测试数据表明，安装opCode加速后，能够显著提升应用性能。

[上一页](#)[下一页](#)

15.3 合并字段缓存

合并字段缓存

[上一页](#)[下一页](#)

默认情况下，字段缓存是自动生成的，在开发完成之后，基本上数据库的变动变得比较少，因此可以考虑合并字段缓存到对应的模型类，这样能够减少每次读取字段缓存的IO开销。

合并的方法是在Runtime/Data/_fields下面找到对应的字段缓存文件，例如，User模型的字段缓存文件中

```
return array (
    0 => 'id',
    1 => 'create_time',
    2 => 'update_time',
    3 => 'status',
    4 => 'account',
    5 => 'password',
    6 => 'nickname',
    7 => 'email',
    8 => 'remark',
    9 => 'avatar',
    '_autoinc' => true,
    '_pk' => 'id',
```

的内容可能是：);

把上面这段代码拷贝到UserModel类的开头，设置为fields属性即可：

为fields属性即可：

```
protected $fields = array (
    0 => 'id',
    1 => 'create_time',
    2 => 'update_time',
    3 => 'status',
    4 => 'account',
    5 => 'password',
    6 => 'nickname',
    7 => 'email',
    8 => 'remark',
    9 => 'avatar',
    10 => 'max_login',
    11 => 'login_count',
    12 => 'last_login_time',
    13 => 'last_login_ip',
    '_autoinc' => true,
    '_pk' => 'id',
);
```

3.1版本以后，字段缓存文件的格式采用JSON格式编码存在，

所以不能直接拷贝，需要按照以上格式自己定义fields属性。

注意：如果在某个模型类中进行了多数据库切换操作，请不要合并。

[上一页](#)[下一页](#)

15.4 优化SQL

优化SQL

[上一页](#) [下一页](#)

通常网站的性能瓶颈在数据库查询，如果你希望你的网站在一定阶段之内保持稳定，优化你的SQL和数据库是非常必要的优化环节。优化数据库是一个很大的话题，这里只是摘要一些比较关键的优化参考建议，并且需要具体分析项目的情况才能给出最合理的优化建议，所以具体的优化建议你应该咨询你公司的架构师或者DBA。

下面是一部分比较重要的建议：

1、选择正确的存储引擎

以 MySQL为例，包括有两个存储引擎 MyISAM 和 InnoDB，每个引擎都有利有弊。

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT() 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。但是它支持“行锁”，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

2、优化字段的数据类型

记住一个原则，越小的列会越快。对于大多数的数据库引擎来说，硬盘操作可能是最重大的瓶颈。所以，把你的数据变得紧凑会对这种情况非常有帮助，因为这减少了对硬盘的访问。

如果一个表只会有几列罢了（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。当然，你也需要留够足够的扩展空间。

3、为搜索字段添加索引

索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你总会经常用来做搜索，那么最好是为其建立索引，除非你要搜索的字段是大的文本字段，那应该建立全文索引。

4、避免使用Select 从数据库里读出越多的数据，那么查询就会变得越慢。并且，如果你的数据库服务器和WEB服务器是两台独立的服务器的话，这还会增加网络传输的负载。即使你要查询数据表的所有字段，也尽量不要用*通配符，善用内置提供的字段排除定义也许能给带来更多的便利。

5、使用 ENUM 而不是 VARCHAR

ENUM 类型是非常快和紧凑的。在实际上，其保存的是 TINYINT，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。例如，性别、民族、部门和状态之类的这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHAR。

6、尽可能的使用 NOT NULL

除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。NULL其实需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。当然，这里并不是说你就不能使用NULL了，现实情况是很复杂的，依然会有些情况下，你需要使用NULL值。

7、固定长度的表会更快

如果表中的所有字段都是“固定长度”的，整个表会被认为是“static”或“fixed-length”。例如，表中没有如下类型的字段：VARCHAR，TEXT，BLOB。只要你包括了其中一个这些字段，那么这个表就不是“固定长度静态表”了，这样，MySQL引擎会用另一种方法来处理。

固定长度的表会提高性能，因为MySQL搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。

并且，固定长度的表也更容易被缓存和重建。不过，唯一的副作用是，固定长度的字段会浪费一些空间，因为定长的字段无论你用不用，他都是要分配那么多的空间。

使用“垂直分割”技术，你可以分割你的表成为两个一个是定长的，一个则是不定长的。

8、垂直分割“垂直分割”是一种把数据库中的表按列变成几张表的方法，这样可以降低表的复杂度和字段的数目，从而达到优化的目的。

例如：在User表中有一个字段是家庭地址，这个字段是可选字段，相比起，而且你在数据库操作的时候除了个人信息外，你并不需要经常读取或是改写这个字段。那么，为什么不把他放到另外一张表中呢？这样会让你的表有更好的性能，大家想想是不是，大量的时候，我对于用户表来说，只有用户ID，用户名，口令，用户角色等会被经常使用。小一点的表总是会有好的性能。

另外，你需要注意的是，这些被分出去的字段所形成的表，你不会经常性地Join他们，不然的话，这样的性能会比不分割时还要差，而且，会是极数级的下降。

9、EXPLAIN 你的 SELECT 查询；

使用 EXPLAIN 关键字可以让你知道MySQL是如何处理你的SQL语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的.....等等，等等。

通常我们可以对比较复杂的尤其是涉及到多表的SELECT语句，把关键字EXPLAIN加到前面。你可以使用phpmyadmin来做这个事。

[上一页](#) [下一页](#)

15.5 替换入口

替换入口

[上一页](#)[下一页](#)

如果正式部署到生产环境后，除非你在入口文件中添加了除常量定义之外的其他代码和逻辑，否则我们建议你用系统编译生成的缓存文件替换入口文件。

[上一页](#)[下一页](#)

15.6 前端优化

前端优化

[上一页](#)[下一页](#)

优化完后端和数据库之后，我们紧接着要做的就是针对输出的页面优化你的前端页面和资源文件，主要包括对图片、JS和样式文件的优化。

我们建议采用下列网页性能测试工具进行检测和分析，会给出相关的优化建议：

| PageSpeed | 谷歌开发的工具，网站管理员和网络开发人员可以使用PageSpeed来评估他们网页的性能，并获得有关如何改进性能的建议。 |

|-----|-----|

| yslow | YSlow可以对网站的页面进行分析，并告诉你为了提高网站性能，如何基于某些规则而进行优化。 |

[上一页](#)[下一页](#)

16. 部署

部署

[上一页](#)[下一页](#)

本章介绍了在部署阶段可能会需要解决的问题。

[上一页](#)[下一页](#)

16.1 PATH_INFO支持

PATH_INFO支持

[上一页](#)[下一页](#)

如果发生在本地测试正常，但是一旦部署到服务器环境后会发生只能访问首页的情况，很有可能是你的服务器或者空间不支持PATH_INFO所致。

新版内置提供了对PATH_INFO的兼容判断处理，但是不能确保在所有的环境下面都可以支持。如果你确认你的空间不支持PATH_INFO的URL方式的话，有下面几种方式可以处理：

修改URL_PATHINFO_FETCH配置参数

新版内置了通过对ORIG_PATH_INFO,REDIRECT_PATH_INFO,REDIRECT_URL三个系统\$_SERVER变量的判断处理来兼容读取\$_SERVER['PATH_INFO']，如果你的主机环境有更特殊的设置，可以修改

URL_PATHINFO_FETCH参数，改成你的环境配置对应的PATH_INFO的系统变量兼容获取名称，例如：

'URL_PATHINFO_FETCH' => 'ORIG_PATH_INFO,REDIRECT_URL,其他参数...'

如果你的环境没有任何对应的系统变量，那么可以封装一个获取方法，例如：

```
function get_path_info(){
    // 根据你的环境兼容获取PATH_INFO 具体代码略
    return $path; // 直接返回获取到的PATH_INFO信息
}
```

然后我们修改下

URL_PATHINFO_FETCH参数的配置值，改为：

'URL_PATHINFO_FETCH' => ':get_path_info' 配置后，系统会自动读取get_path_info方法来获取\$_SERVER['PATH_INFO']的值。

配置你的WEB服务器重写规则模拟PATH_INFO实现

如果你有服务器或者空间的配置权限，可以考虑通过配置URL重写规则来模拟实现。

具体可以参考后面的隐藏index.php中的内容。

采用兼容URL模式运行（这是不得已的方法）

这是最坏的方法，配置你的URL模式为3（表示兼容URL模式）

然后在需要生成URL的地方采用U方法动态生成即可。

[上一页](#)[下一页](#)

16.2 隐藏index.php

隐藏index.php

[上一页](#) [下一页](#)

为了更好的实现SEO优化，我们需要隐藏URL地址中的index.php，由于不同的服务器环境配置方法区别较大，apache环境下面的配置我们可以参考5.9 URL重写来实现，就不再多说了，这里大概说明下IIS和Nginx下面的基本配置方法和思路。

IIS环境

如果你的服务器环境支持ISAPI_Rewrite的话，可以配置httpd.ini文件，添加下面的内容：

RewriteRule (.*)\$ /index\.php?s=\$1 [I] 在IIS的高版本下面可以配置web.Config，在中间添加rewrite节点：

```
<rewrite>
<rules>
<rule name="OrgPage" stopProcessing="true">
<match url="^(.*)$" />
<conditions logicalGrouping="MatchAll">
<add input="{HTTP_HOST}" pattern="^(.*)$" />
<add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
<add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
</conditions>
<action type="Rewrite" url="index.php/{R:1}" />
</rule>
</rules>
</rewrite>
```

Nginx环境

在Nginx低版本中，是不支持PATHINFO的，但是可以通过在Nginx.conf中配置转发规则实现：

```
location / { // ....省略部分代码
    if (!-e $request_filename) {
        rewrite ^(.*)$ /index.php?s=$1 last;
        break;
    }
}
```

其实内部是转发到了ThinkPHP提供的兼容

模式的URL，利用这种方式，可以解决其他不支持PATHINFO的WEB服务器环境。

如果你的ThinkPHP安装在二级目录，Nginx的伪静态方法设置如下，其中youdomain是所在的目录名称。

```
location /youdomain/ {
    if (!-e $request_filename){
        rewrite ^/youdomain/(.*)$ /youdomain/index.php?s=$1 last;
    }
}
```

[上一页](#)[下一页](#)

16.3 二级域名部署

二级域名部署

[上一页](#) [下一页](#)

ThinkPHP支持分组的二级域名部署，该功能可以使项目中的多个分组呈现为二级域名的形式，例如经过配置二级域名部署，可以把：

<http://domain.com/index.php/Admin/>或者<http://domain.com/Admin/>

变为 <http://admin.domain.com/> 访问方式。

先配置域名，以apache为例，配置如下：

```
#主域名      DocumentRoot D:\htdocs\www
              ServerName domain.com      #子域名
              DocumentRoot D:\htdocs\www
              ServerName admin.domain.com
              ServerAlias *.domain.com    然后配置host，以windows为例编辑
```

C:\WINDOWS\system32\drivers\etc\hosts 文件，增加下面两行：

```
127.0.0.1 domain.com
127.0.0.1 admin.domain.com 接下来修改程序的配置文件config.php如下
'APP_GROUP_LIST'      => 'Home,Test,Admin',
'DEFAULT_GROUP'      => 'Home',
'APP_SUB_DOMAIN_DEPLOY'=>1, // 开启子域名配置
/*子域名配置
*格式如：'子域名'=>array('分组名/[模块名]','var1=a&var2=b');
*/
'APP_SUB_DOMAIN_RULES'=>array(
    'admin'=>array('Admin/'), // admin域名指向Admin分组
    'test'=>array('Test/'),  // test域名指向Test分组
),
```

[上一页](#) [下一页](#)

16.4 定制错误页面

定制错误页面

[上一页](#)[下一页](#)

默认情况下，ThinkPHP在发生错误的时候，显示的是系统默认的错误页面，正式上线的时候，为了统一用户体验，我们可以定制自己的错误页面，通常有两种方法：

一、定制系统错误页面模板

系统默认的错误模板位于：ThinkPHP/Tpl/think_exception.tpl

我们只需要在项目中修改TMPL_EXCEPTION_FILE配置参数重新指定错误模板即可。

```
'TMPL_EXCEPTION_FILE'=>'./App/Tpl/Public/error.html' // 定义公共错误模板 注意错误模板的路径是基于入口文件的相对地址或者使用服务器的绝对地址，错误模板中可以使用的变量有：
```

`$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。

二、设置错误重定向页面

如果想网站发生错误的时候重定向到一个指定的URL 而不是读取错误模板，我们还可以直接设置

ERROR_PAGE参数。 `'ERROR_PAGE'=>'/Public/error.html'` // 定义错误跳转页面URL地址 注意ERROR_PAGE所指向的页面不能再使用异常的模板变量了。

[上一页](#)[下一页](#)

16.5 设置时区

设置时区

[上一页](#)[下一页](#)

如果你的服务器分布在不同的地区或者国家，那么有可能有些应用所在的服务器和访问的区域间隔较大，导致服务器时间不准确，我们可以通过设置默认时区的方法来处理。

我们只需要在项目配置文件中添加：

```
'DEFAULT_TIMEZONE'=>'Asia/Singapore' // 设置默认时区为新加坡
```

[上一页](#)[下一页](#)

17. SAE支持

SAE支持

[上一页](#)[下一页](#)

新版提供了SAE的支持，本章主要学习如何基于ThinkPHP进行SAE开发。

[上一页](#)[下一页](#)

17.1 SAE介绍

SAE介绍

[上一页](#)[下一页](#)

Sina App Engine (简称SAE) 是新浪研发中心开发的国内首个公有云计算平台，是新浪云计算战略的核心组成部分，作为一个简单高效的分布式Web服务开发、运行平台越来越受开发者青睐。

SAE环境和普通环境有所不同，它是一个分布式服务器集群，能让你的程序同时运行在多台服务器中。并提供了很多高效的分布式服务。SAE为了提升性能和安全，禁止了本地IO写操作，使用MemcacheX、Storage等存储型服务代替传统IO操作，效率比传统IO读写操作高，有效解决因IO瓶颈导致程序性能低下的问题。

正是因为SAE和普通环境的不同，使得普通程序不能直接放在SAE上，需要经过移植才能放在SAE上运行。也使得很多能在SAE上运行的程序不能在普通环境下运行。

ThinkPHP对SAE平台的支持是采用了引擎扩展的方式，具有自己的独创特性。采用SAE引擎扩展能最大程度的使用ThinkPHP的标准版的特性，让开发人员感受不到SAE和普通环境的差别。甚至可以不学习任何SAE知识，只要会ThinkPHP开发，就能将你的程序运行在SAE上。SAE版ThinkPHP具有以下特性：

横跨性：能让同样的代码既能在SAE环境下运行，也能在普通环境下运行。解决了使用SAE不能在本地调试代码的问题。SAE版ThinkPHP还自带SAE服务模拟功能。用户即使使用了原生的SAE服务

(SaeStorage , SaeRank等) 也能在本地运行。

平滑性：我们还是按照以前一样使用ThinkPHP，但是您已经不知不觉的使用了SAE服务，不用特意学习SAE服务，降低学习成本。比如你不用特意的去学习KVDB服务，你在SAE环境下使用ThinkPHP的F函数就已经使用了KVDB的服务。

完整性：SAE开发下面功能没有任何删减，支持ThinkPHP标准模式的所有功能。甚至在SAE上有些功能还有增强。

大多SAE移植程序都是使用Wrappers实现，SAE版ThinkPHP没有使用Wrappers，使用SAE的原始服务接口，运行效率比用Wrappers更高。

[上一页](#)[下一页](#)

17.2 获取SAE

获取SAE

[上一页](#)[下一页](#)

有两种方式获取和安装SAE的ThinkPHP版本。

SAE官方获取：你可以在SAE的[应用仓库](#)选择ThinkPHP框架的[SAE引擎](#)，点击安装，进行一键安装。在安装过程中SAE会自动为你做好一些列初始化工作。通过SAE官方获得的方法是最简单的，但获取的代码不一定是官方最新的。

ThinkPHP官方获取：可以在官方网站的下载->扩展（<http://thinkphp.cn/down-extend.html>）中下载到最新的ThinkPHP的SAE引擎扩展，或通过SVN下载：

<http://thinkphp.googlecode.com/svn/trunk/Extend/Engine/>

你需要将下载的文件放在ThinkPHP的引擎目录下（ThinkPHP/Extend/Engine）

然后修改你的项目入口文件，把原来的 `require './ThinkPHP/ThinkPHP.php'`；改成 `require './ThinkPHP/Extend/Engine/Sae.php'`；注意，SAE引擎扩展具有横跨性，即使在本地开发也一样可以支持，所以你无需在本地开发的过程中切换不同的入口。接下来，你可以像标准模式的ThinkPHP开发一样进行SAE平台开发了。

在本地开发完成后，上传到SAE平台需要做一些初始化工作，例如初始化Mysql，Memcache，KVDB服务。SAE平台不支持IO写操作，所以你不能在SAE上首次运行入口文件生成项目目录。你可以在本地运行入口文件，本地生成好项目目录后再提交到SAE上。

[上一页](#)[下一页](#)

17.3 SAE开发

SAE开发

[上一页](#) [下一页](#)

ThinkPHP的SAE开发和标准版本的ThinkPHP基本一样，你无需了解SAE的接口用法，ThinkPHP的SAE引擎已经自动为你整合了SAE的接口，只要掌握ThinkPHP开发，你就能轻松掌握基于ThinkPHP的SAE开发。

下面是我们给出的一些利用SAE引擎开发过程的一些注意事项，能够帮助你更好的完成SAE的开发和部署。

配置

SAE引擎运行时拥有SAE自己的惯例配置和专有配置，因此配置文件加载顺序为：

惯例配置->项目配置->SAE惯例配置->SAE专有配置

SAE惯例配置和SAE专有配置中的配置项将会覆盖项目配置。

SAE惯例配置：位于 引擎目录/Sae/Conf/convention_sae.php，其中定义了程序在SAE上运行时固定的数据库连接配置项。

SAE专有配置：位于项目的Conf目录下，文件名为config_sae.php，大家可以将针对SAE的配置写到其中。

注：SAE惯例配置和SAE专有配置是针对SAE环境的独有配置，在本地运行时将不会加载。

数据库

开发者不需要在项目配置文件(config.php)中定义和SAE相关的数据库配置项，只需要定义本地调试时连接的数据库即可。代码提交到SAE时无需修改任何配置项也能运行，因为SAE惯例配置会自动覆盖你的项目配置文件中的数据库配置。

代码在SAE上运行时会进行分布式数据库连接，并读写分离。

缓存

在SAE开发过程中，你仍然可以使用ThinkPHP内置的缓存方法进行处理。下面是SAE引擎使用不同的缓存方法在本地和SAE平台下的区别（注意这个区别SAE引擎会自动判断处理）：

| 缓存方法 | 本地运行 | SAE平台 |

|-----|-----|-----|

| S缓存 | 默认使用File方式实现 | 固定使用Memcache实现，所以在SAE下 DATA_CACHE_TYPE配置项将失效。你如果需要使用SAE提供的Mecache服务，直接使用S函数就可以 |

| F缓存 | 使用File实现 | 使用KVDB实现 |

| 静态缓存 | 生成静态Html文件 | 静态文件存入KVDB中 |

| SQL队列 | 支持File、Xcache和APC方式 | 使用KVDB存储 |

新版的ThinkPHP支持SQL缓存队列功能，我们可以配置DB_SQL_BUILD_CACHE 开启SQL语句解析缓

存。在SAE平台下固定使用KVDB存储SQL缓存，因此DB_SQL_BUILD_QUEUE配置项将不起作用。并且在SAE下运行时会用Counter服务记录SQL缓存队列出队次数，在Counter的管理后台

<http://sae.sina.com.cn/?m=counter>

如果你看到计算器名称为think_queue_out_times 的数值很大，说明你设置的队列个数太小，需要调整DB_SQL_BUILD_LENGTH 配置项。

文件上传

文件上传仍然使用UploadFile扩展类库上传文件，使用方法不变。同样的代码在本地运行时将会上传到指定的目录，在SAE上运行时就会自动使用Storage服务，将文件上传到指定的Storage中。首先你需要在SAE平台上创建一个Storage的domain用于存放上传的文件：

<http://sae.sina.com.cn/?m=storage> 这里可以建立多个domain。而我们的文件会上传到哪个domain，是由上传路径的第一个目录名称决定的。如：

`$upload->savePath = './Public/Uploads/';` 会上传到名为Public的domain。你也不用在这个domain下创建Uploads文件夹，SAE的Storage服务会为你自动创建。

图片地址的问题：

我们使用UploadFile类上传图片，在本地和在SAE下图片的浏览地址是不一样的。比如有张图片地址为"PUBLIC/upload/1.jpg"，PUBLIC 是一个模板替换变量，他会被替换为Public文件夹所在目录的地址，我们可以通过浏览器的查看源代码功能查看被替换后是什么效果。可以看见，替换后为"/Public/upload/1.jpg"。但是在SAE上图片并没有在Public/upload目录下，而是在storage中。我们需要将/Public/替换为storage的域名，在SAE上才能正常显示。

我们在SAE专有配置Conf/config_sae.php文件中 定义如下代码：

```
<?php
return array(
    'TMPL_PARSE_STRING'=>array(
        '/Public/upload'=>sae_storage_root('Public').'/upload'
    )
);
```

这样，在SAE

上会把 /Public/upload 替换为storage的地址，在SAE上图片也能正常显示。

文件删除问题：

因为上传的文件在本地和SAE存放的地方不一样，所以我们不能直接用unlink删除文件。SAE版ThinkPHP新增sae_unlink函数实现兼容。如：`sae_unlink('./Public/Uploads/xxx.jpg');`在本地运行时，会删除Public/Uploads文件夹下的图片。而在SAE上运行时，会删除domain为Public的Storage中的图片。此函数会删除哪个domian的文件也是由路径的第一个目录名称决定的。

图片处理

SAE引擎在图片处理方面也做了自动处理，在本地和SAE平台的区别如下：

| 图片功能 | 本地运行 | SAE平台 |

|-----|-----|-----|

| 缩略图 | 调用Image类库处理 | 自动使用SaeImage服务 |

| 验证码 | 调用Image类库处理 | 自动使用SaeVcode服务 |

你完全不用去学习怎么用SaeImage生成缩略图，也不用学习SaeVcode服务怎么用，你还是按照以前的方

本文档使用 [看云](#) 构建

式使用ThinkPHP进行验证码和缩略图功能就可以了。

使用验证码的时候需要注意，在本地运行时验证码默认为数字形式，而在SAE上运行时验证码为数字+字母形式，而且存在字母大小写问题。如果你希望验证码区分大小写的话，需要将验证码统一转化为大写后进行匹配。

```
if(md5(strtoupper($_POST['verify']))!= $_SESSION['verify']){
    //验证错误处理代码
如: }
```

日志记录

SAE版ThinkPHP同样实现了生成系统日志功能，在本地运行会将日志记录到项目的Runtime/Logs文件夹下，而在SAE上运行会将日志记录到SAE平台的日志中心：

<http://sae.sina.com.cn/?m=applog>

请在搜索框选择中的下拉菜单处选择“debug”进行查看。

Trace信息

建议在开发程序时配置SHOW_PAGE_TRACE=>true 开启页面Trace信息。开启后，代码在SAE环境下运行时显示一些SAE独有的Trace信息，有助于我们开发。你可能会到以下trace信息。

模板缓存：Trace信息名称为 “[SAE]模板缓存”

在SAE下不会将模板编译缓存生成在Runtime目录下，而是存放在Memcache中。如果你想查看模板编译后的缓存，这里显示的就是模板缓存在Memcache中的缓存名称。你可以在SAE的memcache服务管理平台输入缓存名称得到缓存内容：

<http://sae.sina.com.cn/?m=mcmng>

注：你看得缓存内容，都是以一串数字开始，这数字和缓存内容无关，是记录的缓存生成时间。

核心缓存：Trace信息名称为 “[SAE]核心缓存”

它记录的是核心编译缓存在Memcache中的缓存名称。如果你要获得核心编译缓存，比如我们要用核心编译缓存代替入口文件的时候。你可以在SAE的Memcache服务管理平台 输入这里记录的缓存名称获得。

注：

- 在开启调试时不会生成核心编译缓存，如果你获得核心编译缓存，请先关闭调试。
- 缓存内容开头的数字是记录的缓存生成时间，请将数字去掉后再作为入口文件。

静态缓存：Trace信息名称为 “[SAE]静态缓存”

它记录了生成的静态缓存在KVDB中的名称。目前SAE管理平台没有能直接输入KVDB名称获得内容的地方，大家需要自己写程序获取内容。

注：此Trace信息是在生成静态缓存的时候才会出现。如果你访问到的页面没有执行生成静态缓存的操作时，将不会有此条Trace信息。

隐藏index.php

SAE不支持.htaccess文件，但我们可以使用SAE提供的AppConfig服务实现伪静态。

在你项目的根目录建立config.yaml文件，代码为：

```
handle:
- rewrite: if(!is_dir() && !is_file() && path~"^(.*)$") goto "index.php/$1"
```

这样就可以隐藏入口了。

比如这样的地址 <http://serverName/index.php/Blog/read/id/1>也能通过
<http://serverName/Blog/read/id/1>访问。

代码横跨性建议

SAE版ThinkPHP，是具有横跨性的，请不要破坏它的横跨性。比如，不要在项目配置文件中写和SAE数据库相关配置项。自己写代码时，也要尽量做到横跨性，这样就可以让同样的代码既能在SAE下运行，也能在普通环境下运行，使你在本地调试完后上传到SAE也不用修改任何代码就能运行。

下面是一些保持代码横跨性的建议：

(1) 尽量少使用原生的SAE服务

能使用ThinkPHP自带函数替代的，尽量使用ThinkPHP自带函数。比如要使用SAE的KVDB服务，在ThinkPHP中完全可以用F函数代替。如果要使用SAE的Memcache服务，都使用S函数实现。这样就不会导致你的代码从SAE转移到普通环境后性能很低。

个别SAE服务无法使用ThinkPHP自带函数代替的，才考虑使用原生的SAE服务。

(2) 利用IS_SAE常量

ThinkPHP的SAE引擎增加了IS_SAE常量，能判断代码运行环境是普通环境还是SAE环境。如果你有段代码在普通环境和在SAE环境下实现方式不同，你可以使用IS_SAE进行判断后做不同处理或者加载不同的文件。

(3) 利用SAE专有文件

在SAE惯例配置中，我们可以看见除了配置了固定的数据库配置项，还有一个SAE_SPECIALIZED_FILES配置项，它定义了系统专有文件。目前已经定义了UploadFile类和Image类的SAE专有文件，所以当我们的代码 `import("@.ORG.UploadFile")` 在本地运行时按普通方式导入项目下 `Lib/ORG/UploadFile.class.php` 文件，而在SAE上运行是系统检查到 `UploadFile.class.php` 有SAE专有文件，它导入的是SAE_SPECIALIZED_FILES配置项中定义的文件地址。这样实现了普通环境和SAE环境下同样的代码导入了不同类库，而类的调用方法都是一样的，只是现实方法不同，这样就能保证了代码的横跨性。

你也可以自己建立SAE专有文件，你可以将专有文件放在和普通文件同级目录，这样不用定义SAE_SPECIALIZED_FILE配置项，系统也能识别专有文件。比如我们在 `Image.class.php` 的文件的同级目录如果定义了一个名为 `Image_sae.class.php` 的文件，则系统SAE上运行时，导入 `Image.class.php` 文件时会改为导入 `Image_sae.class.php` 文件。

如果一个类库既定义了同级目录下的专有文件，也在SAE_SPECIALIZED_FILE配置项中有定义，则会优先导入同级目录下的专有文件。建议大家如果需要建立专有文件时，在普通文件同级目录下建立。

如果导入的类库没有SAE专有文件，在SAE下运行时也会导入普通文件。

我们可以利用SAE专有文件，针对普通环境和SAE环境封装不同的类库，但类库的使用方法都是相同的，从而让类库的客户端代码具有横跨性。

(4) 利用SAE专有配置

当遇到SAE和普通环境配置需要不一样时，你可以把普通环境的配置写到项目配置文件 `Conf/config.php`

中，而 将SAE需要用的配置写到SAE专有配置Conf/config_sae.php中。

[上一页](#)[下一页](#)

18. REST支持

REST支持

[上一页](#)[下一页](#)

新版增加了对REST的支持，本章主要学习如何基于ThinkPHP进行REST开发。

[上一页](#)[下一页](#)

18.1 REST介绍

REST介绍

[上一页](#)[下一页](#)

REST(Representational State Transfer表述性状态转移)是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性。REST提出了一些设计概念和准则：

- 1、网络上的所有事物都被抽象为资源（ resource ）；
- 2、每个资源对应一个唯一的资源标识（ resource identifier ）；
- 3、通过通用的连接器接口（ generic connector interface ）对资源进行操作；
- 4、对资源的各种操作不会改变资源标识；
- 5、所有的操作都是无状态的（ stateless ）。

需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML以及HTML这些现有的广泛流行的协议和标准。

传统的请求模式和REST模式的请求模式区别：

作用	传统模式	REST模式
列举出所有的用户	GET /users/list	GET /users
列出ID为1的用户信息	GET /users/show/id/1	GET /users/1
插入一个新的用户	POST /users/add	POST /users
更新ID为1的用户信息	POST /users/mdy/id/1	PUT /users/1
删除ID为1的用户	POST /users/delete/id/1	DELETE /users/1

关于更多的REST信息，可以参考：<http://zh.wikipedia.org/wiki/REST>

[上一页](#)[下一页](#)

18.2 REST模式

REST模式

[上一页](#)[下一页](#)

新版增加了Rest模式用于支持RESTFul开发，REST模式主要提供下面的一些功能：

- 路由增加请求类型和资源类型判断支持；
- 支持资源类型自动检测；
- 支持请求类型自动检测；
- RESTFul方法支持；
- 可以设置允许的请求类型列表；
- 可以设置允许请求和输出的资源类型；
- 可以设置默认请求类型和默认资源类型；

要使用REST模式，需要在入口文件中设置

```
define('MODE_NAME', 'rest') // 采用rest模式运行 REST模式更多情况下作为接口应用提供支持，我们来陆续了解下REST模式和标准模式的主要区别和使用。
```

[上一页](#)[下一页](#)

18.3 REST配置

REST配置

[上一页](#)[下一页](#)

Rest模式增加的REST相关配置参数如下：

配置名	说明	默认值
REST_METHOD_LIST	REST允许的请求类型列表	get,post,put,delete
REST_DEFAULT_METHOD	REST默认请求类型	get
REST_CONTENT_TYPE_LIST	REST允许请求的资源类型列表	html,xml,json,rss
REST_DEFAULT_TYPE	REST默认的资源类型	html
REST_OUTPUT_TYPE	REST允许输出的资源类型列表	array('xml' => 'application/xml', 'json' => 'application/json', 'html' => 'text/html',),

这些参数的设置只是rest模式的默认配置，可以在项目配置文件中改变。

[上一页](#)[下一页](#)

18.4 REST路由

REST路由

[上一页](#)[下一页](#)

Rest模式下面的路由定义必须用数组方式，并且规则调整为：

内部路由：`array('路由规则或者正则','路由地址','路由额外参数','请求类型','资源类型')` 外部路由：`array('路由规则或者正则','外部地址','重定向代码','请求类型','资源类型')` 主要区别是增加了请求类型和资源类型定义，提交类型包括GET POST DELETE PUT，不区分大小写，资源类型是指访问URL地址的资源后缀，允许多个资源类型，用逗号分隔多个，例如：

```
array('info/:id\d','Info/read_html','','get','html')
array('info/:id\d','Info/read_xml','','get','xml,rss')
array('info/:id\d','Info/insert','','post','html')
array('info/:id\d','Info/update','','put','html')
array('info/:id\d','Info/delete','','delete','html')    所有
```

<http://serverName.com/info/3> 的URL访问 根据不同的请求类型和资源类型会被路由到Info模块的相关RESTful操作方法。

[上一页](#)[下一页](#)

18.5 REST方法

REST方法

[上一页](#)[下一页](#)

RESTFul方法和标准模式的操作方法定义主要区别在于，需要对请求类型和资源类型进行判断，大多数情况下，通过路由定义可以把操作方法绑定到某个请求类型和资源类型。如果你没有定义路由的话，需要自

```
Class InfoAction extends Action {
  Public function rest() {
    switch ($this->_method){
      case 'get': // get请求处理代码
        if ($this->_type == 'html'){
        }elseif($this->_type == 'xml'){
        }
        break;
      case 'put': // put请求处理代码
        break;
      case 'post': // put请求处理代码
        break;
    }
  }
}
```

已在操作方法里面添加判断代码，示例： } 在Rest操作方法中，可以使用\$this->_type获取当前访问的资源类型，用\$this->_method获取当前的请求类型。

REST模式的Action类还提供了response方法用于REST输出：

response输出数据	
用法	response(\$data,\$type="", \$code=200)
参数	data（必须）：要输出的数据 type（可选）：要输出的类型，支持REST_OUTPUT_TYPE参数允许的类型，如果为空则取REST_DEFAULT_TYPE参数设置值 code（可选）：HTTP状态
返回值	无

Response方法会自动对data数据进行输出类型编码，目前支持的包括xml json html。
除了普通方式定义Restful操作方法外，系统还支持另外一种自动调用方式，就是根据当前请求类型和资源类型自动调用相关操作方法。系统的自动调用规则是：

| 定义规范 | 说明 |

|-----|-----|

| 操作名_提交类型_资源后缀 | 标准的Restful方法定义，例如 read_get_pdf |

| 操作名_资源后缀 | 当前提交类型和REST_DEFAULT_METHOD相同的时候，例如read_pdf |

| 操作名_提交类型 | 当前资源后缀和REST_DEFAULT_TYPE相同的时候，例如read_post |

要使用这种方式的前提就是不能为当前操作定义方法，这样在空操作的检查之前系统会首先按照上面的定义规范顺序检查是否存在方法定义，如果检测到相关的restful方法则不再检查后面的方法规范，例如我们

```
Class InfoAction extends Action {
  Public function read_get_xml(){
    // 输出id为1的Info的XML数据
  }
  Public function read_xml(){
    // 输出id为1的Info的XML数据
  }
  Public function read_json(){
    // 输出id为1的Info的json数据
  }
}
```

定义了InfoAction如下：

项目配置中设置了如下rest相

```
'REST_METHOD_LIST'      =>'get,post,put', // 允许的请求类型列表
'REST_DEFAULT_METHOD'    =>'get', // 默认请求类型
'REST_DEFAULT_TYPE'      =>'html', // 默认的资源类型
```

关参数：'REST_CONTENT_TYPE_LIST' =>'html,xml,json', // REST允许请求的资源类型列表

如果我们访问的URL是：`http://www.domain.com/Info/read/id/1.xml` 假设我们没有定义路

由，这样访问的是Info模块的read操作，那么上面的请求会调用InfoAction类的 `read_get_xml`方法，而

不是`read_xml`方法，但是如果访问的URL是：`http://www.domain.com/Info/read/id/1.json`

那么则会调用`read_json`方法。

[上一页](#)[下一页](#)

19. 杂项

杂项

[上一页](#)[下一页](#)

本章有很多内容可能涉及到扩展类库中的功能活着需要额外的扩展支持，请确保你已经下载相关的扩展。

[上一页](#)[下一页](#)

19.1 Session支持

Session支持

[上一页](#)[下一页](#)

系统提供了Session管理和操作的完善支持，全部操作可以通过一个内置的session函数完成。

Session 用于Session 设置、获取、删除和管理操作	
用法	session(\$name, \$value="")
参数	name（必须）：如果传入数组 则表示进行session初始化，如果传入null表示清空当前session，如果是字符串则表示session赋值、获取或者操作。 Value（可选）：要设置的session值，如果传入null表示删除session，默认为空字符串
返回值	见详（根据具体的用法返回不同的值）

session函数是一个多元化操作函数，传入不同的参数调用可以完成不同的功能操作，包括下面一些功能。session初始化设置

如果session方法的名字参数传入数组则表示进行session初始化设置，例如：

```
session(array('name'=>'session_id','expire'=>3600));
```

支持传入的session参数包括：

参数名	说明
id	session_id值
name	session_name 值
path	session_save_path 值
prefix	session 本地化空间前缀
expire	session.gc_maxlifetime 设置值
domain	session.cookie_domain 设置值
use_cookies	session.use_cookies 设置值
use_trans_sid	session.use_trans_sid 设置值
type	session handler类型，可以使用handler驱动扩展

Session初始化设置方法 无需手动调用，在App类的初始化工作结束后会自动调用，通常项目只需要配置SESSION_OPTIONS参数即可，SESSION_OPTIONS参数的设置是一个数组，支持的索引名和前面的session初始化参数相同。

默认情况下，初始化之后系统会自动启动session，如果不希望系统自动启动session的话，可以设置SESSION_AUTO_START为false，例如：`'SESSION_AUTO_START' =>false` 关闭自动启动后可以项

目的公共文件或者在控制器中通过手动调用`session_start`或者`session(['start'])` 启动session。session赋值

Session赋值比较简单，直接使用：`session('name', 'value');` //设置session session取值

Session取值使用：`$value = session('name');` session删除

`session('name', null);` // 删除name 要删除所有的session，可以使用：

`session(null);` // 清空当前的session session判断

要判断一个session值是否已经设置，可以使用 `session('?name');` 用于判断名称为name的session值是否已经设置session管理

session方法支持一些简单的session管理操作，用法如下：`session('[操作名]');` 支持的操作名包括：

| 操作名 | 含义 |

|-----|-----|

| start | 启动session |

| pause | 暂停session写入 |

| destroy | 销毁session |

| regenerate | 重新生成session id |

`session('[pause]');` // 暂停session写入

`session('[start]');` // 启动session

`session('[destroy]');` // 销毁session

使用示例如下：`session('[regenerate]');` // 重新生成session id 本地化支持

如果在初始化session设置的时候传入prefix参数或者单独设置了SESSION_PREFIX参数的话，就可以启用本地化session管理支持。启动本地化session后，所有的赋值、取值、删除以及判断操作都会自动支持本地化session。

本地化session支持开启后，生成的session数据格式由原来的

`$_SESSION['name']` 变成 `$_SESSION['前缀'] ['name']` session handler支持

是初始化session设置的时候，如果传入了type参数，则会自动引入对应的handler驱动，驱动扩展目录位于Extend/Driver/Session目录下面（详见扩展部分）。

[上一页](#) [下一页](#)

19.2 Cookie支持

Cookie支持

[上一页](#)[下一页](#)

系统内置了一个cookie函数用于支持和简化Cookie的相关操作。

Cookie 用于Cookie 设置、获取、删除操作	
用法	cookie(\$name, \$value="", \$option=null)
参数	name (必须) : 要操作的cookie变量 value (可选) : 要设置的cookie值 option (可选) : 传入的cookie设置参数, 默认为空
返回值	见详 (根据具体的用法返回不同的值)

cookie函数是一个多元化操作函数，同一个函数的不同参数调用方式可以完成cookie的设置、获取和删除

```
cookie('name','value'); //设置cookie
操作。Cookie设置 cookie('name','value',3600); // 指定cookie保存时间 还可以支持参数传入的方式完成复杂的cookie赋值，下面是对cookie的值设置3600秒有效期，并且加上cookie前缀
think`cookie('name','value',array('expire'=>3600,'prefix'=>'think'))
```

数组参数可以采用query形式参数 cookie('name','value','expire=3600&prefix=think_') 和上面的用法等效。

传入的option参数支持prefix,expire,path,domain四个索引参数，如果没有传入或者传入空值的则获取cookie很简单，无论是怎么设置的cookie，只需要使用：

```
$value = cookie('name'); 如果没有设置cookie前缀的话 相当于 $value = $_COOKIE['name']
如果设置了cookie前缀的话，相当于 $value = $_COOKIE['前缀+name']
```

```
**Cookie删除**
删除某个cookie的值，使用： cookie('name',null); 要删除所有的Cookie值，可以使用
cookie(null); // 清空当前设定前缀的所有cookie值
cookie(null,'think'); // 清空指定前缀的所有cookie值
```

3.1版本开始，cookie方法增加对数组的支持，采用轻量级的json编码格式保存 减少存储空间。例如：

```
cookie('name',array('name1','name2'));
```

[上一页](#)[下一页](#)

19.3 日期和时间

日期和时间

[上一页](#) [下一页](#)

扩展类库ORG.Util.Date类提供了时间和日期的操作功能，目前提供的方法包括：

isLeapYear判断是否是闰年	
用法	isLeapYear(\$year="")
参数	year（可选）：年，留空取实例化Date类的时候生成的year属性
返回值	布尔值

dateDiff和实例化生成的日期计算日期差	
用法	dateDiff(\$date,\$elaps= "d")
参数	date（必须）：要比较的日期 elaps（可选）：比较的跨度，默认为d，支持 Y-年M-月w-星期d-天h-小时m-分钟s-秒
返回值	数字

timeDiff个性化的计算日期差	
用法	timeDiff(\$time,\$precision=false)
参数	time（必须）：要比较的时间 precision（可选）：返回的精度
返回值	字符串

firstDayOfMonth计算月份的第一天	
用法	firstDayOfMonth()
参数	无
返回值	Date对象可直接输出

firstDayOfYear计算年的第一天	
用法	firstDayOfYear()
参数	无
返回值	Date对象可直接输出

lastDayOfMonth计算月份的最后一天	
用法	lastDayOfMonth()
参数	无
返回值	Date对象可直接输出

lastDayOfYear计算年份的最后一天	
用法	lastDayOfYear()
参数	无
返回值	Date对象可直接输出

maxDayOfMonth计算月份的最大天数	
用法	maxDayOfMonth()
参数	无
返回值	数字

dateAdd取得指定间隔日期	
用法	dateAdd(\$number=0,\$interval= "d")
参数	<p>number (可选) : 间隔数目, 默认为0</p> <p>interval (可选) : 间隔类型, 默认为d, 支持 :</p> <p>yyyy年</p> <p>q-季度</p> <p>m-月</p> <p>y-dayofyear</p> <p>d-日</p> <p>w-周</p> <p>ww-weekofyear</p> <p>h-小时</p> <p>n-分钟</p> <p>s-秒</p>
返回值	Date对象

numberToCh日期数字转中文，用于日和月、周	
用法	numberToCh(\$number)
参数	number（必须）：日期数字
返回值	字符串

yearToCh年份数字转中文	
用法	yearToCh(\$yearStr,\$flag=false)
参数	yearStr（必须）：年份字 flag（可选）：是否显示公元
返回值	字符串

magicInfo判断日期所属干支生肖星座	
用法	magicInfo(\$type)
参数	type（必须）：获取信息类型，支持 XZ星座GZ干支SX生肖
返回值	字符串

要使用Date类，首先需要实例化，使用示例：

```
import('ORG.Util.Date');// 导入日期类
$Date = new Date('2012-03-25');
$Date->isLeapYear(); // 判断是否闰年
echo $Date->dateDiff('2020-03-25','m'); // 比较日期差
echo $Date->lastDayOfMonth(); // 计算当月的最后一天
echo $Date->maxDayOfMonth(); // 计算当月的最大天数
```

[上一页](#)[下一页](#)

19.4 WML开发

WML开发

[上一页](#)[下一页](#)

ThinkPHP不仅仅只是支持Html网站开发，如果我们的项目需要实现WAP支持，我们只需要做如下步骤就可以支持WML开发：

一、设置模板后缀为.wml

在项目配置里面增加配置参数 `'TMPL_TEMPLATE_SUFFIX'=>'.wml'` 并确保你的所有模板文件都以wml后缀保存即可。并注意模板wml文件的定义规范~

二、更改模板输出类型

系统默认的模板输出类型是HTML，也就是text/html，所以我们需要修改为WML的输出类型

```
'TMPL_CONTENT_TYPE'=>'text/vnd.wap.wml' 如果你的网站编码不是UTF-8的话，还需要设置  
'DEFAULT_CHARSET'=>'gbk'
```

[上一页](#)[下一页](#)

19.5 多语言

多语言

[上一页](#)[下一页](#)

ThinkPHP内置多语言支持，如果你的应用涉及到国际化的支持，那么可以定义相关的语言包文件。任何字符串形式的输出，都可以定义语言常量。要启用多语言功能，需要配置开启多语言行为（确保你下载的是完整版本，如果不是需要单独下载多语言检测行为扩展），在项目的配置目录下面的行为定义文件

```
return array(
    // 添加下面一行定义即可
    'app_begin' => array('CheckLang')
);
```

tags.php中，添加：表示在app_begin标签位置执行多语言检测行为。

要开启语言包功能，需要开启 'LANG_SWITCH_ON' => true, // 开启语言包功能 其他的配置参数

```
'LANG_AUTO_DETECT' => true, // 自动侦测语言 开启多语言功能后有效
'LANG_LIST'          => 'zh-cn', // 允许切换的语言列表 用逗号分隔
'VAR_LANGUAGE'       => 'l', // 默认语言切换变量
```

包括：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。

具体的语言包文件命名规范如下：

项目公共语言包	语言目录/common.php
项目分组语言包	语言目录/分组名.php
项目模块语言包	不存在分组情况：语言目录/模块名（小写）.php
	存在分组的情况：语言目录/分组名/模块名（小写）.php

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。

分组的模块语言包定义受TMPL_FILE_DEPR参数配置影响，如果你修改了TMPL_FILE_DEPR参数，例如：

'TMPL_FILE_DEPR'=>'_'，那么，分组的模块语言包定义方式应该改为：

语言目录/分组名_模块名（小写）.php语言文件定义

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

ThinkPHP语言文件定义采用返回数组方式：); 要在程

```
L('define2','语言定义');
```

序里面设置语言定义的值，使用下面的方式：\$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

[上一页](#)[下一页](#)

19.6 数据分页

数据分页

[上一页](#)[下一页](#)

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，下面是数据分页的两种示例。

第一种：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,
$this->assign('list',$list);// 赋值数据集
$this->assign('page',$show);// 赋值分页输出
$this->display(); // 输出模板
```

第二种：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)
$this->assign('list',$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。例本文档使用 [看云](#) 构建

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。

[上一页](#)[下一页](#)

19.7 文件上传

文件上传

[上一页](#)[下一页](#)

上传类使用ORG.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：

```
<input type="file" name="photo1">
<input type="file" name="photo2">
<input type="file" name="photo3">
```

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->getErrMsg());
}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、uniq:
| 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[]'>
```

识别。

[上一页](#)[下一页](#)

19.8 验证码

验证码

[上一页](#)[下一页](#)

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

```
Public function verify(){
    import('ORG.Util.Image');
    Image::buildImageVerify();
}
```

类中增加一个verify方法来用于显示验证码：} Image类的
buildImageVerify方法用于生成验证码，

buildImageVerify 生成图像验证码		
用法	buildImageVerify(\$length,\$mode,\$type,\$width,\$height,\$verifyName)	
用法	length	验证码的长度，默认为4位数
	model	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
	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：
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：

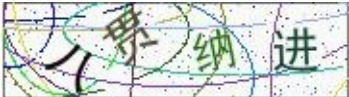
GBVerify生成中文验证码	
用法	GBVerify (\$length,\$type,\$width,\$height,\$fontface,\$verifyName)

参数	<p>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
</p>
----	---

```
Public function verify(){
    import("ORG.Util.Image");
    Image::GBVerify();
}
```

用法示例：

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；

[上一页](#)[下一页](#)

19.9 图片添加水印

图片添加水印

[上一页](#)[下一页](#)

可以通过使用Image类的水印方法给图片添加水印支持，例如：

water 给图片添加水印	
用法	water(\$source, \$water, \$savename=null, \$alpha=80)
参数	source（必须）：原图文件名。 Water（必须）：水印图片文件名 savename（可选）：要保存的图片名，如果留空则用source alpha（可选）：水印图片的alpha值，默认为80，范围为0~100
返回值	无

```
import('ORG.Util.Image');
$image = new Image();
// 给avator.jpg 图片添加logo水印
使用示例： $image->water('/avator.jpg', '/logo.jpg');
```

[上一页](#)[下一页](#)

19.10 IP获取和定位

IP获取和定位

[上一页](#)[下一页](#)

系统内置了get_client_ip方法用于获取客户端的IP地址，使用示例：`$ip = get_client_ip();`；如果要支持IP定位功能，需要使用扩展类库ORG.Net.IpLocation，并且要配合IP地址库文件一起使用，例如：

```
import('ORG.Net.IpLocation');// 导入IpLocation类
$Ip = new IpLocation('UTFWry.dat');// 实例化类 参数表示IP地址库文件
$area = $Ip->getlocation('203.34.5.66');// 获取某个IP地址所在的位置 如果传入的参数为空，则会自动获取当前的客户端IP地址，要正确输出位置，必须配合UTF8编码的ip地址库文件，否则可能还需要进行编码转换。IP地址库文件和IpLocation类库位于同一目录即可。
```

[上一页](#)[下一页](#)

19.11Thinkphp的 I 方法

ThinkPHP函数详解：I方法

概述

正如你所见到的一样，I方法是ThinkPHP众多单字母函数中的新成员，其命名来自于英文Input（输入），主要用于更加方便和安全的获取系统输入变量，可以用于任何地方，用法格式如下：

I('变量类型.变量名','默认值','过滤方法')

变量类型是指请求方式或者输入类型，包括：

变量类型 含义

get 获取GET参数

post 获取POST参数

param 自动判断请求类型获取GET、POST或者PUT参数

request 获取REQUEST 参数

put 获取PUT 参数

session 获取 \$_SESSION 参数

cookie 获取 \$_COOKIE 参数

server 获取 \$_SERVER 参数

globals 获取 \$GLOBALS参数

注意：变量类型不区分大小写。

变量名则严格区分大小写。

默认值和过滤方法均属于可选参数。

用法

我们以GET变量类型为例，说明下I方法的使用：

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

复制代码

支持默认值：

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

复制代码

采用方法过滤：

```
echo I('get.name','','htmlspecialchars'); // 采用htmlspecialchars方法对$_GET['name']
```

进行过滤，如果不存在则返回空字符串

复制代码

支持直接获取整个变量类型，例如：

```
I('get.');// 获取整个$_GET 数组
```

用同样的方式，我们可以获取post或者其他输入类型的变量，例如：

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

复制代码

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

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

复制代码

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

并且param类型变量还可以用数字索引的方式获取URL参数（必须是PATHINFO模式参数有效，无论是GET还是POST方式都有效），例如：

当前访问URL地址是

<http://serverName/index.php/New/2013/06/01>

复制代码

那么我们可以通过

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

复制代码

事实上，param变量类型的写法可以简化为：

I('id');// 等同于 I('param.id')

I('name');// 等同于 I('param.name')

本文档使用 [看云](#) 构建

复制代码

变量过滤

使用I方法的时候 变量其实经过了两道过滤，首先是全局的过滤，全局过滤是通过配置VAR_FILTERS参数，这里一定要注意，3.1版本之后，VAR_FILTERS参数的过滤机制已经更改为采用array_walk_recursive方法递归过滤了，主要对过滤方法的要求是必须引用返回，所以这里设置htmlspecialchars是无效的，你可以自定义一个方法，例如：

```
function filter_default(&$value){
    $value = htmlspecialchars($value);
}
```

复制代码

然后配置：

'VAR_FILTERS'=>'filter_default'

复制代码

如果需要进行多次过滤，可以用：

'VAR_FILTERS'=>'filter_default,filter_exp'

复制代码

filter_exp方法是框架内置的安全过滤方法，用于防止利用模型的EXP功能进行注入攻击。

因为VAR_FILTERS参数设置的是全局过滤机制，而且采用的是递归过滤，对效率有所影响，所以，我们更建议直接对获取变量过滤的方式，除了在I方法的第三个参数设置过滤方法外，还可以采用配置DEFAULT_FILTER参数的方式设置过滤，事实上，该参数的默认设置是：

'DEFAULT_FILTER' => 'htmlspecialchars'

复制代码

也就是说，I方法的所有获取变量都会进行htmlspecialchars过滤，那么：

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

同样，该参数也可以支持多个过滤，例如：

'DEFAULT_FILTER' => 'strip_tags,htmlspecialchars'

复制代码

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

复制代码

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

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

复制代码

I方法的第三个参数如果传入函数名，则表示调用该函数对变量进行过滤并返回（在变量是数组的情况下自
本文档使用 [看云](#) 构建

动使用array_map进行过滤处理），否则会调用PHP内置的filter_var方法进行过滤处理，例如：

```
I('post.email','',FILTER_VALIDATE_EMAIL);
```

表示 会对\$_POST['email'] 进行 格式验证，如果不符合要求的话，返回空字符串。

（关于更多的验证格式，可以参考 官方手册的filter_var用法。）

或者可以用下面的字符标识方式：

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

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

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

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

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

一旦过滤参数设置为NULL，即表示不再进行任何的过滤。

20. 附录

附录

[上一页](#)[下一页](#)

[上一页](#)[下一页](#)

20.1 常量参考

常量参考

[上一页](#)[下一页](#)

预定义常量

常量	说明
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
THINK_VERSION	框架版本号
THINK_RELEASE	框架发行日期

这些预定义常量不会随着环境的变化而变化。

路径常量

系统和项目的路径常量用于系统默认的目录规范，可以通过重新定义改变，如果不希望定制目录，这些常量一般不需要更改。

| 常量名 | 说明 | 默认值 |

|-----|-----|-----|

| CORE_PATH | 系统核心类库目录 | THINK_PATH.'Lib/' |

| EXTEND_PATH | 系统扩展目录 | THINK_PATH.'Extend/' |

| MODE_PATH | 模式扩展目录 | EXTEND_PATH.'Mode/' |

| ENGINE_PATH | 引擎扩展目录 | EXTEND_PATH.'Engine/' |

| VENDOR_PATH | 第三方类库目录 | EXTEND_PATH.'Vendor/' |

| LIBRARY_PATH | 系统扩展类库目录 | EXTEND_PATH.'Library/' |

| COMMON_PATH | 项目公共目录 | APP_PATH.'Common/' |

| LIB_PATH | 项目类库目录 | APP_PATH.'Lib/' |

| RUNTIME_PATH | 项目运行时目录 | APP_PATH.'Runtime/' |

| CONF_PATH | 项目配置目录 | APP_PATH.'Conf/' |

| LOG_PATH | 项目日志目录 | RUNTIME_PATH.'Logs/' |

CACHE_PATH	项目模板缓存目录	RUNTIME_PATH.'Cache/'
LANG_PATH	项目语言包目录	APP_PATH.'Lang/'
TEMP_PATH	项目缓存目录	RUNTIME_PATH.'Temp/'
DATA_PATH	项目数据目录	RUNTIME_PATH.'Data/'
TMPL_PATH	项目模板目录	APP_PATH.'Tpl/'
HTML_PATH	项目静态缓存目录	APP_PATH.'Html/'

系统常量

下面这些系统常量会随着开发环境的改变或者设置的改变而产生变化。

常量名	说明
IS_CGI	是否属于 CGI 模式
IS_WIN	是否属于 Windows 环境
IS_CLI	是否属于命令行模式
ROOT	网站根目录地址
APP	当前项目（入口文件）地址
GROUP	当前分组的 URL 地址
URL	当前模块的 URL 地址
ACTION	当前操作的 URL 地址
SELF	当前 URL 地址
INFO	当前的 PATH_INFO 字符串
EXT	当前 URL 地址的扩展名
APP_NAME	当前项目名
GROUP_NAME	当前分组名
MODULE_NAME	当前模块名
ACTION_NAME	当前操作名
APP_DEBUG	是否开启调试模式
MODE_NAME	当前模式名称
APP_PATH	当前项目路径
THINK_PATH	系统框架路径
MEMORY_LIMIT_ON	系统内存统计支持
RUNTIME_FILE	项目编译缓存文件名
THEME_NAME	当前主题名称
THEME_PATH	当前模板主题路径
APP_TMPL_PATH	当前模板 URL 路径
LANG_SET	当前浏览器语言
MAGIC_QUOTES_GPC	MAGIC_QUOTES_GPC
NOW_TIME	当前请求时间（3.1 新增）
IS_GET	当前是否 GET 请求（3.1 新增）

IS_POST	当前是否POST请求（3.1新增）
IS_PUT	当前是否PUT请求（3.1新增）
IS_DELETE	当前是否DELETE请求（3.1新增）
IS AJAX	当前是否AJAX请求（3.1新增）

[上一页](#)[下一页](#)

20.2 配置参考

配置参考

[上一页](#)[下一页](#)

这里列出了系统的惯例配置和内置系统行为的配置参数列表。

惯例配置

配置名	说明	默认值
应用设置		
APP_STATUS	应用调试模式状态 调试模式开启后有效 默认为debug 可扩展 并自动加载对应的配置文件	debug
APP_FILE_CASE	是否检查文件的大小写 对Windows 平台有效	false
APP_AUTOLOAD_PATH	自动加载机制的自动搜索路径,注意搜索顺序	
APP_TAGS_ON	系统标签扩展开关	true
APP_SUB_DOMAIN_DEPLOY	是否开启子域名部署	false
APP_SUB_DOMAIN_RULES	子域名部署规则	array()
APP_SUB_DOMAIN_DENY	子域名禁用列表	array()
APP_GROUP_LIST	项目分组设定,多个组之间用逗号分隔	
ACTION_SUFFIX	操作方法后缀	
默认值设置		
DEFAULT_APP	默认项目名称, @表示当前项目	@
DEFAULT_LANG	默认语言	zh-cn
DEFAULT_THEME	默认模板主题名称	
DEFAULT_GROUP	默认分组名	Home
DEFAULT_MODULE	默认模块名	Index
DEFAULT_ACTION	默认操作名	index
DEFAULT_CHARSET	默认输出编码	utf-8
DEFAULT_TIMEZONE	默认时区	PRC
DEFAULT AJAX_RETURN	默认AJAX 数据返回格式,可选JSON XML	JSON
DEFAULT_FILTER	默认参数过滤方法	htmlspecialchars

COOKIE_EXPIRE	Coodie有效期（秒）	3600
COOKIE_DOMAIN	Cookie有效域名	
COOKIE_PATH	Cookie路径	/
COOKIE_PREFIX	Cookie前缀 避免冲突	
数据库配置		
DB_TYPE	数据库类型	mysql
DB_DSN	数据库连接信息DSN串	
DB_HOST	数据库服务器地址	localhost
DB_NAME	数据库名称	
DB_USER	数据库用户名	root
DB_PWD	数据库用户密码	
DB_PORT	数据库端口	
DB_FIELDS_CACHE	是否开启数据表字段缓存	true
DB_FIELDTYPE_CHECK	是否开启字段类型检查	false
DB_CHARSET	数据库编码	utf8
DB_DEPLOY_TYPE	数据库部署方式 0 集中式 1 分布式	0
DB_RW_SEPARATE	数据库是否需要读写分离 分布式部署下有效	false
DB_MASTER_NUM	设置读写分离后 主服务器数量	1
DB_SLAVE_NO	设置读写分离后 指定从服务器序号（3.1新增）	
DB_SQL_BUILD_CACHE	数据库查询的SQL创建缓存	false
DB_SQL_BUILD_QUEUE	SQL缓存队列的缓存方式	file
DB_SQL_BUILD_LENGTH	SQL缓存的队列长度	20
DB_SQL_LOG	是否开启SQL日志记录（3.1新增）	false
数据缓存设置		
DATA_CACHE_TIME	数据缓存有效期 0表示永久缓存	0
DATA_CACHE_COMPRESS	数据缓存是否压缩缓存	false
DATA_CACHE_CHECK	数据缓存是否校验缓存	false
DATA_CACHE_TYPE	数据缓存类型	File
DATA_CACHE_PATH	缓存路径设置 (仅对File方式缓存有效)	TEMP_PATH
DATA_CACHE_SUBDIR	使用子目录缓存(仅对File方式缓存有效)	false
	子目录缓存级别(仅对File方式缓存有	

	效)	
错误设置		
ERROR_MESSAGE	错误显示信息，部署模式有效	
ERROR_PAGE	错误定向页面，部署模式有效	
SHOW_ERROR_MSG	是否显示错误信息	False
日志设置		
LOG_RECORD	是否记录日志信息	false
LOG_TYPE	默认日志记录类型 0 系统 1 邮件 3 文件 4 SAPI	3
LOG_DEST	日志记录目标	
LOG_EXTRA	日志记录额外信息	
LOG_LEVEL	允许记录的日志级别	EMERG,ALERT,CRIT,ERR
LOG_FILE_SIZE	日志文件大小限制（字节 文件方式有效）	2097152
LOG_EXCEPTION_RECORD	是否记录异常信息日志	false
SESSION设置		
SESSION_AUTO_START	是否自动开启Session	true
SESSION_OPTIONS	session 配置数组	array()
SESSION_TYPE	session handler类型	
SESSION_PREFIX	session 前缀	
VAR_SESSION_ID	sessionID的提交变量	session_id
模板引擎设置		
TMPL_CONTENT_TYPE	默认模板输出类型	text/html
TMPL_ACTION_ERROR	默认错误跳转对应的模板文件	系统模板目录下的 dispatch_jump.tpl
TMPL_ACTION_SUCCESS	默认成功跳转对应的模板文件	同上
TMPL_EXCEPTION_FILE	异常页面的模板文件	系统模板目录下的 think_exception.tpl
TMPL_DETECT_THEME	自动侦测模板主题	false
TMPL_TEMPLATE_SUFFIX	默认模板文件后缀	.html
TMPL_FILE_DEPR	模板文件模块与操作之间的分割符，只对项目分组部署有效	/
URL设置		
URL_CASE_INSENSITIVE	URL是否不区分大小写	false
	URL访问模式支持 0 (普通模式); 1 (PATHINFO 模式);	

	2 (REWRITE 模式); 3 (兼容模式)	
URL_PATHINFO_DEPR	PATHINFO模式下的参数分割符	/
URL_PATHINFO_FETCH	用于兼容判断PATH_INFO 参数的 SERVER替代变量列表	ORIG_PATH_INFO REDIRECT_PATH_INFO REDIRECT_URL
URL_HTML_SUFFIX	URL伪静态后缀设置	
URL_404_REDIRECT	404跳转页面 部署模式有效 (3.1新增)	
URL_PARAMS_BIND	URL变量绑定到Action方法参数 (3.1新增)	true
系统变量名称设置		
VAR_GROUP	默认分组获取变量	g
VAR_MODULE	默认模块获取变量	m
VAR_ACTION	默认操作获取变量	a
VAR_AJAX_SUBMIT	默认的AJAX提交变量	ajax
VAR_TEMPLATE	默认模板主题切换变量	t
VAR_PATHINFO	兼容模式获取变量	s
VAR_URL_PARAMS	PATHINFOURL参数变量	_URL_
VAR_FILTERS	全局系统变量的默认过滤方法 多个用 逗号分割 (3.1新增)	
OUTPUT_ENCODE	是否开启页面压缩输出 (3.1新增)	true

行为配置

这里仅仅列出了系统内置的行为扩展的配置参数

配置名	说明	默认值
CheckRoute行为配置		
URL_ROUTER_ON	是否开启URL路由	false
URL_ROUTE_RULES	默认路由规则	array()
ContentReplace行为配置		
TMPL_PARSE_STRING	模板替换规则	array()
ParseTemplate行为配置		
TMPL_ENGINE_TYPE	默认模板引擎	Think
TMPL_CACHFILE_SUFFIX	默认模板缓存后缀	.php
TMPL_DENY_FUNC_LIST	模板引擎禁用函数	echo,exit
TMPL_DENY_PHP	是否禁用PHP原生代码	false

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	}
TAGLIB_BEGIN	标签库标签开始标记	<
TAGLIB_END	标签库标签结束标记	>
TAGLIB_LOAD	是否使用内置标签库之外的其它标签库，默认自动检测	true
TAGLIB_BUILD_IN	内置标签库名称	cx
TAGLIB_PRE_LOAD	需要预先加载的标签库	
TMPL_VAR_IDENTIFY	模板变量识别。留空自动判断	array
TMPL_STRIP_SPACE	是否去除模板文件里面的html空格与换行	true
TMPL_CACHE_ON	是否开启模板编译缓存	true
TMPL_CACHE_TIME	模板缓存有效期 0为永久	0
LAYOUT_ON	是否启用布局	false
LAYOUT_NAME	当前布局名称	layout
TMPL_LAYOUT_ITEM	布局模板的内容替换标识	{CONTENT}
ReadHtmlCache行为配置		
HTML_CACHE_ON	是否开启静态缓存	false
HTML_CACHE_RULES	静态缓存规则	array()
HTML_CACHE_TIME	静态缓存有效期（秒）	60
HTML_FILE_SUFFIX	静态缓存后缀	.html
ShowPageTrace行为配置		
SHOW_PAGE_TRACE	显示页面Trace信息	false
ShowRuntime行为配置		
SHOW_RUN_TIME	是否显示运行时间	false
SHOW_ADV_TIME	是否显示详细的运行时间	false
SHOW_DB_TIMES	是否显示数据库查询和写入次数	false
SHOW_CACHE_TIMES	是否显示缓存操作次数	false
SHOW_USE_MEM	是否显示内存开销	false
SHOW_LOAD_FILE	是否显示加载文件数	false
SHOW_FUN_TIMES	是否显示函数调用次数	false
TokenBuild行为配置		
TOKEN_ON	是否开启令牌验证	true
TOKEN_NAME	令牌验证的表单隐藏字段名称	hash
TOKEN_TYPE	令牌验证哈希规则	md5

[上一页](#)[下一页](#)

20.3 关于升级

关于升级

[上一页](#)[下一页](#)

参考官方提供的升级指导手册。

[上一页](#)[下一页](#)

20.4 大事记

大事记

[上一页](#)[下一页](#)

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

2006-02-01（元宵节）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

2010-10-01 ThinkPHP 发布2.1RC1版本

2011-05-01 ThinkPHP 发布2.1正式版本

2012-01-15 ThinkPHP 发布2.2正式版本和3.0RC1版本

2012-02-07 ThinkPHP发布3.0RC2版本

2012-03-05 ThinkPHP 发布3.0正式版本

2012-10-08 ThinkPHP 发布3.1正式版本

2012-11-15 ThinkPHP 发布3.1.2版本

2013-01-15 ThinkPHP 7周年纪念

[上一页](#)[下一页](#)

鸣谢

鸣谢

[上一页](#)[下一页](#)

在ThinkPHP的开发和本手册的编写过程中，要感谢ThinkPHP文档小组成员、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！

[上一页](#)[下一页](#)

关于

关于

[上一页](#)

本手册通过在线手册<http://doc.thinkphp.cn/manual.html>]
开放api获取内容编译生成，因此内容和在线手册一致。欢迎大家下载学习！

官方会定期将在线手册编译成CHM版本供大家下载，敬请关注官网资讯。

版本信息

编译时间：2013-01-15 14:15:10

下载地址：<http://www.thinkphp.cn/down/267.html>

BUG反馈：<http://www.thinkphp.cn/bug/index.html>

更新日志

[2013-01-10]

- 新增内容头部标题显示
- 新增了前后翻页按钮，让阅读变得更方便
- 改进图片显示，图片直接抓取到本地并编译到CHM中，解决了原来相对路径图片不显示的BUG，实现了整个手册完全离线

[2013-01-04]

- 修复了搜索列表显示乱码的BUG
- 修复了Linux核心系统下目录显示乱码的BUG

[2012-12-19]

- 完成第一次编译，发布下载

[上一页](#)

