



为API开发
而生

@官方文档小组

ThinkPHP V5

完全开发手册

目 录

序言

基础

安装ThinkPHP

开发规范

目录结构

架构

架构总览

生命周期

入口文件

URL访问

模块设计

命名空间

自动加载

Traits引入

API友好

多层MVC

配置

配置格式

配置加载

读取配置

动态配置

独立配置

配置作用域

环境变量配置

路由

路由模式

路由定义

批量注册

变量规则

组合变量

路由参数

路由地址

资源路由

快捷路由

路由别名

路由分组

MISS路由

闭包支持

路由绑定

绑定模型

域名路由

URL生成

控制器

控制器定义

控制器初始化

前置操作

跳转和重定向

空操作

空控制器

多级控制器

分层控制器

Rest控制器

自动定位控制器

资源控制器

请求

请求信息

输入变量

更改变量

请求类型

请求伪装

方法参数绑定

注入请求对象

HTTP头信息

方法注入

属性注入

伪静态

数据库

连接数据库

基本使用

查询构造器

查询数据

添加数据

更新数据

删除数据

查询方法

查询语法

链式操作

where

table

alias
field
order
limit
page
group
having
join
union
distinct
lock
cache
comment
fetchSql
force
using
bind
partition
strict
failException
sequence

聚合查询

时间查询

高级查询

视图查询

子查询

原生查询

事务操作

监听SQL

存储过程

数据集

分布式数据库

模型

定义

模型初始化

新增

更新

删除

查询

聚合

- 获取器
- 修改器
- 时间戳
- 软删除
- 类型转换
- 数据完成
- 查询范围
- 模型分层
- 数组访问和转换
- JSON序列化
- 事件
- 关联
 - 一对一关联
 - 一对多关联
 - 远程一对多
 - 多对多关联
 - 动态属性
 - 关联预载入

- 聚合模型

视图

- 视图实例化
- 模板引擎
- 模板赋值
- 模板渲染
- 输出替换

模板

- 模板定位
- 模板标签
- 变量输出
- 系统变量
- 请求参数
- 使用函数
- 使用默认值
- 使用运算符
- 三元运算
- 原样输出
- 模板注释
- 模板布局
- 模板继承
- 包含文件

标签库

内置标签

循环输出标签

比较标签

条件判断

资源文件加载

标签嵌套

原生PHP

定义标签

日志

介绍

日志驱动

日志写入

独立日志

日志清空

写入授权

错误和调试

调试模式

异常处理

抛出异常

Trace调试

变量调试

性能调试

SQL调试

远程调试

404页面

验证

验证器

验证规则

错误信息

验证场景

控制器验证

模型验证

内置规则

静态调用

表单令牌

杂项

缓存

Session

Cookie

- 多语言
- 分页
- 上传
- 验证码
- 图像处理
- 文件处理
- 单元测试

扩展

- 函数
- 类库
- 行为
- 驱动
- Composer包
 - Time
- SAE
- 标签扩展

命令行

- 自动生成目录结构
- 创建类库文件
- 生成类库映射文件
- 生成路由缓存
- 清除缓存文件
- 生成配置缓存文件

MongoDb

- 安装

部署

- 虚拟主机环境
- Linux 主机环境
- URL重写

附录

- 配置参考
- 常量参考
- 助手函数
- 升级指导
- 更新日志

序言

手册阅读须知：本手册仅针对ThinkPHP5.0版本（使用左右键（<-- 和 -->）翻页阅读）



ThinkPHP V5.0——为API开发而设计的高性能框架



V5.0版本由七牛云独家赞助发布



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，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。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。

ThinkPHP5.0版本是一个颠覆和重构版本，采用全新的架构思想，引入了更多的PHP新特性，优化了核心，减少了依赖，实现了真正的惰性加载，支持composer，并针对API开发做了大量的优化，包括路由、日志、异常、模型、数据库、模板引擎和验证等模块都已经重构，不适合原有3.2项目的升级，请慎重考虑商业项目升级，但绝对是新项目的首选（无论是WEB还是API开发）。

主要特性：

规范：遵循PSR-2、PSR-4规范，Composer及单元测试支持；

严谨：异常严谨的错误检测和安全机制，详细的日志信息，为你的开发保驾护航；

灵活：减少核心依赖，扩展更灵活、方便，支持命令行指令扩展；

API友好：出色的性能和REST支持、远程调试，更好的支持API开发；

高效：惰性加载，及路由、配置和自动加载的缓存机制；

ORM：重构的数据库、模型及关联，MongoDb支持；

支持ThinkPHP5的用户请到 [Github](#) 给我们一个 star ^_^

本手册不能替代教程，新手建议先阅读官方的 《5.0快速入门》

ThinkPHP V5.0 官方权威QQ群

新手一群（272433397 开放制）允许扯淡（已满员）

新手二群（369126686 开放制）允许扯淡（已满员）

高级群（50546480 收费制）禁止闲聊

专家群（416914496 邀请制）比较安静

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 thinkphp@qq.com。

对ThinkPHP有任何疑问或者建议，请进入官方讨论区 [<http://www.thinkphp.cn/topic>] 发布相关讨论。

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://www.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。

捐赠我们

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

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

用手机扫描进行**支付宝捐赠**



推荐阅读

《[ThinkPHP5.0快速入门](#)》是官方出品的学习和掌握 `ThinkPHP5.0` 不可多得的入门指引教程，针对新手用户由浅入深给出了详尽的使用。

本系列围绕WEB开发和API开发常用的一系列基础功能进行循序渐进的讲解。



基础

[安装ThinkPHP](#)

[开发规范](#)

[目录结构](#)

安装ThinkPHP

ThinkPHP5 的环境要求如下：

- PHP >= 5.4.0
- PDO PHP Extension
- MBstring PHP Extension
- CURL PHP Extension

严格来说，ThinkPHP 无需安装过程，这里所说的安装其实就是把 ThinkPHP 框架放入 WEB 运行环境（**前提是你的WEB运行环境已经OK**），可以通过下面几种方式获取和安装ThinkPHP。

一、官网下载安装

获取 ThinkPHP 的方式很多，官方网站（<http://thinkphp.cn>）提供了[稳定版本](#)或者带扩展完整版本的下载。

官网的下载版本不一定是最新版本，GIT版本获取的才是保持更新的版本。

二、Composer安装

ThinkPHP5 支持使用 Composer 安装，如果还没有安装 Composer，你可以按 [Composer安装](#) 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

在 Windows 中，你需要下载并运行 [Composer-Setup.exe](#)。

如果遇到任何问题或者想更深入地学习 Composer，请参考 [Composer 文档（英文）](#)，[Composer 中文](#)。

如果你已经安装有 Composer 请确保使用的是最新版本，你可以用 `composer self-update` 命令更新 Composer 为最新版本。

然后在命令行下面，切换到你的web根目录下面并执行下面的命令：

```
composer create-project tophink/think tp5 --prefer-dist
```

如果出现错误提示，请根据提示操作或者参考[Composer中文文档](#)。

如果国内访问 [composer](#) 的速度比较慢，可以参考这里的说明使用国内镜像

三、Git安装

如果你不太了解 **Composer** 或者觉得 **Composer** 太慢，也可以使用 **git** 版本库安装和更新，**ThinkPHP5.0** 拆分为多个仓库，主要包括：

- 应用项目：`https://github.com/top-think/think`
- 核心框架：`https://github.com/top-think/framework`

之所以设计为应用和核心仓库分离，是为了支持 **Composer** 单独更新核心框架。

首先克隆下载应用项目仓库

```
git clone https://github.com/top-think/think tp5
```

然后切换到 **tp5** 目录下面，再克隆核心框架仓库：

```
git clone https://github.com/top-think/framework thinkphp
```

两个仓库克隆完成后，就完成了 **ThinkPHP5.0** 的 **Git** 方式下载，如果需要更新核心框架的时候，只需要切换到thinkphp核心目录下面，然后执行：

```
git pull https://github.com/top-think/framework
```

如果不熟悉 **git** 命令行，可以使用任何一个GIT客户端进行操作，在此不再详细说明。

无论你采用什么方式获取的 **ThinkPHP** 框架，现在只需要做最后一步来验证是否正常运行。

在浏览器中输入地址：

```
http://localhost/tp5/public/
```

如果浏览器输出如图所示：



ThinkPHP V5

十年磨一剑 - 为API开发设计的高性能框架

[V5.0 版本由 [七牛云](#) 独家赞助发布]

恭喜你，现在已经完成 ThinkPHP5 的安装！

如果你无法正常运行并显示 ThinkPHP 的欢迎页面，那么请检查下你的服务器环境：

- PHP 5.4 以上版本（**注意：PHP5.4dev版本和PHP6均不支持**）
- WEB服务器是否正常启动

[推广] [1元享Azure云试用](#)，15分钟轻松建站

开发规范

命名规范

ThinkPHP5 遵循 PSR-2 命名规范和 PSR-4 自动加载规范，并且注意如下规范：

目录和文件

- 目录不强制规范，驼峰及小写+下划线模式均支持；
- 类库、函数文件统一以 `.php` 为后缀；
- 类的文件名均以命名空间定义，并且命名空间的路径和类库文件所在路径一致；
- 类名和类文件名保持一致，统一采用驼峰法命名（首字母大写）；

函数和类、属性命名

- 类的命名采用驼峰法，并且首字母大写，例如 `User`、`UserType`，默认不需要添加后缀，例如 `UserController` 应该直接命名为 `User`；
- 函数的命名使用小写字母和下划线（小写字母开头）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，例如 `getUserName`；
- 属性的命名使用驼峰法，并且首字母小写，例如 `tableName`、`instance`；
- 以双下划线“`__`”打头的函数或方法作为魔法方法，例如 `__call` 和 `__autoload`；

常量和配置

- 常量以大写字母和下划线命名，例如 `APP_PATH` 和 `THINK_PATH`；
- 配置参数以小写字母和下划线命名，例如 `url_route_on` 和 `url_convert`；

数据表和字段

- 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，不建议使用驼峰和中文作为数据表字段命名。

应用类库命名空间规范

应用类库的根命名空间统一为 `app`（可以设置 `app_namespace` 配置参数更改）；

例如：`app\index\controller\Index` 和 `app\index\model\User`。

请避免使用PHP保留字（保留字列表参见 <http://php.net/manual/zh/reserved.keywords.php>）作为常量、类名和方法名，以及命名空间的命名，否则会造成系统错误。

目录结构

下载最新版框架后，解压缩到web目录下面，可以看到初始的目录结构如下：

| | |
|----------------|--------------------------|
| project | 应用部署目录 |
| application | 应用目录（可设置） |
| common | 公共模块目录（可更改） |
| index | 模块目录（可更改） |
| config.php | 模块配置文件 |
| common.php | 模块函数文件 |
| controller | 控制器目录 |
| model | 模型目录 |
| view | 视图目录 |
| ... | 更多类库目录 |
| command.php | 命令行工具配置文件 |
| common.php | 应用公共（函数）文件 |
| config.php | 应用（公共）配置文件 |
| database.php | 数据库配置文件 |
| tags.php | 应用行为扩展定义文件 |
| route.php | 路由配置文件 |
| extend | 扩展类库目录（可定义） |
| public | WEB 部署目录（对外访问目录） |
| static | 静态资源存放目录(css, js, image) |
| index.php | 应用入口文件 |
| router.php | 快速测试文件 |
| .htaccess | 用于 apache 的重写 |
| runtime | 应用的运行时目录（可写，可设置） |
| vendor | 第三方类库目录（Composer） |
| thinkphp | 框架系统目录 |
| lang | 语言包目录 |
| library | 框架核心类库目录 |
| think | Think 类库包目录 |
| traits | 系统 Traits 目录 |
| tpl | 系统模板目录 |
| .htaccess | 用于 apache 的重写 |
| .travis.yml | CI 定义文件 |
| base.php | 基础定义文件 |
| composer.json | composer 定义文件 |
| console.php | 控制台入口文件 |
| convention.php | 惯例配置文件 |
| helper.php | 助手函数文件（可选） |
| LICENSE.txt | 授权说明文件 |
| phpunit.xml | 单元测试配置文件 |
| README.md | README 文件 |
| start.php | 框架引导文件 |
| build.php | 自动生成定义文件（参考） |
| composer.json | composer 定义文件 |
| LICENSE.txt | 授权说明文件 |
| README.md | README 文件 |
| think | 命令行入口文件 |

5.0的部署建议是 `public` 目录作为web目录访问内容，其它都是web目录之外，当然，你必须要修改 `public/index.php` 中的相关路径。如果没法做到这点，请记得设置目录的访问权限或者添加目录列表的保护文件。

router.php用于php自带webserver支持，可用于快速测试
启动命令：`php -S localhost:8888 router.php`

5.0版本自带了一个完整的应用目录结构和默认的应用入口文件，开发人员可以在这个基础之上灵活调整。

上面的目录结构和名称是可以改变的，尤其是应用的目录结构，这取决于你的入口文件和配置参数。
由于ThinkPHP5.0的架构设计对模块的目录结构保留了很多的灵活性，尤其是对于用于存储的目录具有高度的定制化，因此上述的目录结构仅供建议参考。

架构

- 架构总览
- 生命周期
- 入口文件
- URL访问
- 模块设计
- 命名空间
- 自动加载
- Traits引入
- API友好
- 多层MVC

架构总览

ThinkPHP5.0 应用基于 MVC（模型-视图-控制器）的方式来组织。

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。

5.0的URL访问受路由决定，如果关闭路由或者没有匹配路由的情况下，则是基于：

`http://serverName/index.php（或其它应用入口文件）/模块/控制器/操作/参数/值...`

下面的一些概念有必要做下了解，可能在后面的内容中经常会被提及。

入口文件

用户请求的PHP文件，负责处理一个请求（注意，不一定是URL请求）的生命周期，最常见的入口文件就是 `index.php`，有时候也会为了某些特殊的需求而增加新的入口文件，例如给后台模块单独设置的一个入口文件 `admin.php` 或者一个控制器程序入口 `think` 都属于入口文件。

应用

应用在 ThinkPHP 中是一个管理系统架构及生命周期的对象，由系统的 `\think\App` 类完成，应用通常在入口文件中被调用和执行，具有相同的应用目录（`APP_PATH`）的应用我们认为是同一个应用，但一个应用可能存在多个入口文件。

应用具有自己独立的配置文件、公共（函数）文件。

模块

一个典型的应用是由多个模块组成的，这些模块通常都是应用目录下面的一个子目录，每个模块都有自己独立的配置文件、公共文件和类库文件。

5.0支持单一模块架构设计，如果你的应用下面只有一个模块，那么这个模块的子目录可以省略，并且在应用配置文件中修改：

```
'app_multi_module' => false,
```

控制器

每个模块拥有独立的 MVC 类库及配置文件，一个模块下面有多个控制器负责响应请求，而每个控制器其实就是一个独立的控制器类。

控制器主要负责请求的接收，并调用相关的模型处理，并最终通过视图输出。严格来说，控制器不应该过多的介入业务逻辑处理。

事实上，5.0中控制器是可以被跳过的，通过路由我们可以直接把请求调度到某个模型或者其他的类进行处理。

5.0的控制器类比较灵活，可以无需继承任何基础类库。

一个典型的 **Index** 控制器类如下：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'hello,thinkphp!';
    }
}
```

操作

一个控制器包含多个操作（方法），操作方法是URL访问的最小单元。

下面是一个典型的 **Index** 控制器的操作方法定义，包含了两个操作方法：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'index';
    }

    public function hello($name)
    {
        return 'Hello, '.$name;
    }
}
```

操作方法可以不使用任何参数，如果定义了一个非可选参数，则该参数必须通过用户请求传入，如果是URL请求，则通常是 `$_GET` 或者 `$_POST` 方式传入。

模型

模型类通常完成实际的业务逻辑和数据封装，并返回和格式无关的数据。

模型类并不一定要访问数据库，而且在5.0的架构设计中，只有进行实际的数据库查询操作的时候，才会进行数据库的连接，是真正的惰性连接。

ThinkPHP的模型层支持多层设计，你可以对模型层进行更细化的设计和分工，例如把模型层分为逻辑层/服务层/事件层等等。

视图

控制器调用模型类后返回的数据通过视图组装成不同格式的输出。视图根据不同的需求，来决定调用模板引擎进行内容解析后输出还是直接输出。

视图通常会有一系列的模板文件对应不同的控制器和操作方法，并且支持动态设置模板目录。

驱动

系统很多的组件都采用驱动式设计，从而可以更灵活的扩展，驱动类的位置默认是放入核心类库目录下面，也可以重新定义驱动类库的命名空间而改变驱动的文件位置。

行为

行为（Behavior）是在预先定义好的一个应用位置执行的一些操作。类似于 AOP 编程中的“切面”的概念，给某一个切面绑定相关行为就成了一种类 AOP 编程的思想。所以，行为通常是和某个位置相关，行为的执行时间依赖于绑定到了哪个位置上。

要执行行为，首先要在应用程序中进行行为侦听，例如：

```
// 在app_init位置侦听行为
\think\Hook::listen('app_init');
```

然后对某个位置进行行为绑定：

```
// 绑定行为到app_init位置
\think\Hook::add('app_init', '\app\index\behavior\Test');
```

一个位置上如果绑定了多个行为的，按照绑定的顺序依次执行，除非遇到中断。

命名空间

ThinkPHP5 采用了 PHP 的命名空间进行类库文件的设计和规划，并且符合 PSR-4 的自动加载规范。

生命周期

本篇内容我们对ThinkPHP 5.0 的应用请求的生命周期做大致的介绍，以便于开发者了解整个执行流程。

1、入口文件

用户发起的请求都会经过应用的入口文件，通常是 `public/index.php` 文件。当然，你也可以更改或者增加新的入口文件。

通常入口文件的代码都比较简单，一个普通的入口文件代码如下：

```
// 应用入口文件

// 定义项目路径
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
```

一般入口文件以定义一些常量为主，支持的常量请参考后续的内容或者附录部分。

通常，我们不建议在应用入口文件中加入过多的代码，尤其是和业务逻辑相关的代码。

2、引导文件

接下来就是执行框架的引导文件，`start.php` 文件就是系统默认的一个引导文件。在引导文件中，会依次执行下面操作：

- 加载系统常量定义；
- 加载环境变量定义文件；
- 注册自动加载机制；
- 注册错误和异常处理机制；
- 加载惯例配置文件；
- 执行应用；

`start.php` 引导文件首先会调用 `base.php` 基础引导文件，某些特殊需求下面可能直接在入口文件中引入基础引导文件。

如果在你的应用入口文件中更改了默认的引导文件，则上述执行流程可能会跟随发生变化。

3、注册自动加载

系统会调用 `Loader::register()` 方法注册自动加载，在这一步完成后，所有符合规范的类库（包括 `Composer` 依赖加载的第三方类库）都将自动加载。

系统的自动加载由下面主要部分组成：

1. 注册系统的自动加载方法 `\think\Loader::autoload`
2. 注册系统命名空间定义
3. 加载类库映射文件（如果存在）
4. 如果存在 `Composer` 安装，则注册 `Composer` 自动加载
5. 注册 `extend` 扩展目录

一个类库的自动加载检测顺序为：

1. 是否定义类库映射；
2. `PSR-4` 自动加载检测；
3. `PSR-0` 自动加载检测；

可以看到，定义类库映射的方式是最高效的。

4、注册错误和异常机制

执行 `Error::register()` 注册错误和异常处理机制。

由三部分组成：

- 应用关闭方法：`think\Error::appShutdown`
- 错误处理方法：`think\Error::appError`
- 异常处理方法：`think\Error::appException`

注册应用关闭方法是为了便于拦截一些系统错误。

在整个应用请求的生命周期过程中，如果抛出了异常或者严重错误，均会导致应用提前结束，并响应输出异常和错误信息。

5、应用初始化

执行应用的第一步操作就是对应用进行初始化，包括：

- 加载应用（公共）配置；
- 加载扩展配置文件（由 `extra_config_list` 定义）；
- 加载应用状态配置；
- 加载别名定义；
- 加载行为定义；
- 加载公共（函数）文件；
- 注册应用命名空间；
- 加载扩展函数文件（由 `extra_file_list` 定义）；
- 设置默认时区；
- 加载系统语言包；

6、URL访问检测

应用初始化完成后，就会进行URL的访问检测，包括 `PATH_INFO` 检测和URL后缀检测。

5.0的URL访问必须是 `PATH_INFO` 方式（包括兼容方式）的URL地址，例如：

```
http://serverName/index.php/index/index/hello/val/value
```

所以，如果你的环境只能支持普通方式的URL参数访问，那么必须使用

```
http://serverName/index.php?s=/index/index/hello&val=value
```

如果是命令行下面访问入口文件的话，则通过

```
$php index.php index/index/hello/val/value...
```

获取到正常的 `$_SERVER['PATH_INFO']` 参数后才能继续。

7、路由检测

如果开启了 `url_route_on` 参数的话，会首先进行URL的路由检测。

如果一旦检测到匹配的路由，根据定义的路由地址会注册到相应的URL调度。

5.0的路由地址支持如下方式：

- 路由到模块/控制器/操作；
- 路由到外部重定向地址；
- 路由到控制器方法；
- 路由到闭包函数；
- 路由到类的方法；

路由地址可能会受域名绑定的影响。

如果关闭路由或者路由检测无效则进行默认的**模块/控制器/操作**的分析识别。

如果在应用初始化的时候指定了应用调度方式，那么路由检测是可选的。

可以使用 `\think\App::dispatch()` 进行应用调度，例如：

```
App::dispatch(['type' => 'module', 'module' => 'index/index']);
```

8、分发请求

在完成了URL检测和路由检测之后，路由器会分发请求到对应的路由地址，这也是应用请求的生命周期中最重要的一个环节。

在这一步骤中，完成应用的业务逻辑及数据返回。

建议统一使用 `return` 返回数据，而不是 `echo` 输出，如非必要，请不要使用 `exit` 或者 `die` 中断执行。

直接 `echo` 输出的数据将无法进行自动转换响应输出的便利。

下面是系统支持的分发请求机制，可以根据情况选择：

模块/控制器/操作

这是默认的分发请求机制，系统会根据URL或者路由地址来判断当前请求的模块、控制器和操作名，并自动调用相应的访问控制器类，执行操作对应的方法。

该机制下面，首先会判断当前模块，并进行模块的初始化操作（和应用的初始化操作类似），模块的配置参数会覆盖应用的尚未生效的配置参数。

支持模块映射、URL参数绑定到方法，以及操作绑定到类等一些功能。

控制器方法

和前一种方式类似，只是无需判断模块、控制器和操作，直接分发请求到一个指定的控制器类的方法，因此没有进行模块的初始化操作。

外部重定向

可以直接分发请求到一个外部的重定向地址，支持指定重定向代码，默认为301重定向。

闭包函数

路由地址定义的时候可以直接采用闭包函数，完成一些相对简单的逻辑操作和输出。

类的方法

除了以上方式外，还支持分发请求到类的方法，包括：

静态方法：`'blog/:id'=>'\\org\\util\\Blog::read'`

类的方法：`'blog/:id'=>'\\app\\index\\controller\\Blog@read'`

9、响应输出

控制器的所有操作方法都是 `return` 返回而不是直接输出，系统会调用 `Response::send` 方法将最终的应用返回的数据输出到页面或者客户端，并自动转换成 `default_return_type` 参数配置的格式。所以，应用执行的数据输出只需要返回一个正常的PHP数据即可。

10、应用结束

事实上，在应用的数据响应输出之后，应用并没真正的结束，系统会在应用输出或者中断后进行日志保存写入操作。

系统的日志包括用户调试输出的和系统自动生成的日志，统一会在应用结束的时候进行写入操作。

而日志的写入操作受日志初始化的影响。

入口文件

ThinkPHP采用**单一入口模式**进行项目部署和访问，无论完成什么功能，一个应用都有一个统一（但不一定是唯一）的入口。

应该说，所有应用都是从入口文件开始的，并且不同应用的入口文件是类似的。

入口文件定义

入口文件主要完成：

- 定义框架路径、项目路径（可选）
- 定义系统相关常量（可选）
- 载入框架入口文件（必须）

5.0默认的应用入口文件位于 `public/index.php`，内容如下：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
```

入口文件位置的设计是为了让应用部署更安全，`public` 目录为web可访问目录，其他的文件都可以放到非WEB访问目录下面。

修改入口文件位置请查看章节<[部署-虚拟主机环境](#)>

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

给 `APP_PATH` 定义绝对路径会提高系统的加载效率。

在有些情况下，你可能需要加载框架的基础引导文件 `base.php`，该引导文件和 `start.php` 的区别是不会主动执行应用，而是需要自己进行应用执行，下面是一个例子：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架基础引导文件
require __DIR__ . '/../thinkphp/base.php';
// 添加额外的代码
// ...
// 执行应用
\think\App::run()->send();
```


URL访问

URL设计

ThinkPHP 5.0 在没有启用路由的情况下典型的URL访问规则是：

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

支持切换到命令行访问，如果切换到命令行模式下面的访问规则是：

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

可以看到，无论是URL访问还是命令行访问，都采用 `PATH_INFO` 访问地址，其中 `PATH_INFO` 的分隔符是可以设置的。

注意：5.0 取消了URL模式的概念，并且**普通模式的URL访问不再支持**，但参数可以支持普通方式传值，例如：

```
>php.exe index.php(或者其它应用入口文件) 模块/控制器/操作?参数名=参数值&...
```

如果不支持PATHINFO的服务器可以使用兼容模式访问如下：

```
http://serverName/index.php ( 或者其它应用入口文件 ) ?s=/模块/控制器/操作/[参数名/参数值...]
```

必要的时候，我们可以通过某种方式，省略URL里面的模块和控制器。

URL大小写

默认情况下，URL 是不区分大小写的，也就是说 URL 里面的**模块/控制器/操作名**会自动转换为小写，控制器在最后调用的时候会转换为驼峰法处理。

例如：

```
http://localhost/index.php/Index/Blog/read  
// 和下面的访问是等效的  
http://localhost/index.php/index/blog/read
```

如果访问下面的地址

```
http://localhost/index.php/Index/BlogTest/read  
// 和下面的访问是等效的  
http://localhost/index.php/index/blogtest/read
```

在这种URL不区分大小写情况下，如果要访问驼峰法的控制器类，则需要使用：

```
http://localhost/index.php/Index/blog_test/read
```

模块名和操作名会直接转换为小写处理。

如果希望 URL 访问严格区分大小写，可以在应用配置文件中设置：

```
// 关闭URL中控制器和操作名的自动转换
'url_convert' => false,
```

一旦关闭自动转换，URL地址中的控制器名就变成大小写敏感了，例如前面的访问地址就要写成：

```
http://localhost/index.php/Index/BlogTest/read
```

但是下面的URL访问依然是有效的：

```
http://localhost/index.php/Index/blog_test/read
```

下面的URL访问则无效：

```
http://localhost/index.php/Index/blogtest/read
```

需要注意：路由规则中定义的路由地址是按照控制器名的实际名称定义（区分大小写）。

隐藏入口文件

在ThinkPHP5.0中，出于优化的URL访问原则，还支持通过URL重写隐藏入口文件，下面以 **Apache** 为例说明隐藏应用入口文件index.php的设置。

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、在应用入口文件同级目录添加 **.htaccess** 文件，内容如下：

```
<IfModule mod_rewrite.c>
Options +FollowSymlinks -Multiviews
RewriteEngine on

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


模块设计

5.0版本对模块的功能做了灵活设计，默认采用多模块的架构，并且支持单一模块设计，所有模块的命名空间均以 `app` 作为根命名空间（可配置更改）。

目录结构

标准的应用和模块目录结构如下：

| | |
|--------------|------------|
| application | 应用目录（可设置） |
| common | 公共模块目录（可选） |
| common.php | 公共函数文件 |
| route.php | 路由配置文件 |
| database.php | 数据库配置文件 |
| config.php | 应用配置文件 |
| module1 | 模块1目录 |
| config.php | 模块配置文件 |
| common.php | 模块函数文件 |
| controller | 控制器目录 |
| model | 模型目录 |
| view | 视图目录 |
| ... | 更多类库目录 |
| module2 | 模块2目录 |
| config.php | 模块配置文件 |
| common.php | 模块函数文件 |
| controller | 控制器目录 |
| model | 模型目录 |
| view | 视图目录 |
| ... | 更多类库目录 |

遵循ThinkPHP 5.0 的命名规范，模块目录全部采用**小写和下划线**命名。

模块名称请避免使用PHP保留关键字（保留字列表参见 <http://php.net/manual/zh/reserved.keywords.php> ），否则会造成系统错误。

其中 `common` 模块是一个特殊的模块，默认是禁止直接访问的，一般用于放置一些公共的类库用于其他模块的继承。

模块类库

一个模块下面的类库文件的命名空间统一以 `app\模块名` 开头，例如：

```
// index模块的Index控制器类
app\index\controller\Index
// index模块的User模型类
app\index\model\User
```

其中 `app` 可以通过定义的方式更改，例如我们在应用配置文件中修改：

```
'app_namespace' => 'application',
```

那么，index模块的类库命名空间则变成：

```
// index模块的Index控制器类
application\index\controller\Index
// index模块的User模型类
application\index\model\User
```

更多的关于类库和命名空间的关系可以参考下一章节：命名空间。

模块和控制器隐藏

由于默认是采用多模块的支持，所以多个模块的情况下必须在URL地址中标识当前模块，如果只有一个模块的话，可以进行模块绑定，方法是应用的入口文件中添加如下代码：

```
// 绑定当前访问到index模块
define('BIND_MODULE', 'index');
```

绑定后，我们的URL访问地址则变成：

[http://serverName/index.php/控制器/操作/\[参数名/参数值...\]](http://serverName/index.php/控制器/操作/[参数名/参数值...])

访问的模块是 `index` 模块。

如果你的应用比较简单，模块和控制器都只有一个，那么可以在应用公共文件中绑定模块和控制器，如下：

```
// 绑定当前访问到index模块的index控制器
define('BIND_MODULE', 'index/index');
```

设置后，我们的URL访问地址则变成：

[http://serverName/index.php/操作/\[参数名/参数值...\]](http://serverName/index.php/操作/[参数名/参数值...])

访问的模块是 `index` 模块，控制器是 `Index` 控制器。

单一模块

如果你的应用比较简单，只有唯一一个模块，那么可以进一步简化成使用单一模块结构，方法如下：

首先应用配置文件中定义：

```
// 关闭多模块设计
'app_multi_module' => false,
```

然后，调整应用目录的结构为如下：

| | |
|---------------|-----------|
| —application | 应用目录（可设置） |
| —controller | 控制器目录 |
| —model | 模型目录 |
| —view | 视图目录 |
| —... | 更多类库目录 |
| —common.php | 函数文件 |
| —route.php | 路由配置文件 |
| —database.php | 数据库配置文件 |
| —config.php | 配置文件 |

URL访问地址变成

`http://serverName/index.php (或者其它应用入口) /控制器/操作/[参数名/参数值...]`

同时，单一模块设计下的应用类库的命名空间也有所调整，例如：

原来的

```
app\index\controller\Index
app\index\model\User
```

变成

```
app\controller\Index
app\model\User
```

更多的URL简化和定制还可以通过URL路由功能实现。

命名空间

命名空间

ThinkPHP5 采用命名空间方式定义和自动加载类库文件，有效的解决了多模块和 **Composer** 类库之间的命名空间冲突问题，并且实现了更加高效的类库自动加载机制。

如果不清楚命名空间的基本概念，可以参考PHP手册：[PHP命名空间](#)

特别注意的是，如果你需要调用PHP内置的类库，或者第三方没有使用命名空间的类库，记得在实例化类库的时候加上 `\`，例如：

```
// 错误的用法
$class = new stdClass();
$xml = new SimpleXmlElement($xmlstr);
// 正确的用法
$class = new \stdClass();
$xml = new \SimpleXmlElement($xmlstr);
```

在ThinkPHP 5.0 中，只需要给类库正确定义所在的命名空间，并且命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载，从而实现真正的惰性加载。

例如，`\think\cache\driver\File` 类的定义为：

```
namespace think\cache\driver;

class File
{
}
```

如果我们实例化该类的话，应该是：

```
$class = new \think\cache\driver\File();
```

系统会自动加载该类对应路径的类文件，其所在的路径是 `thinkphp/library/think/cache/driver/File.php`。

5.0默认的目录规范是小写，类文件命名是驼峰法，并且首字母大写。

原则上，可以支持驼峰法命名的目录，只要命名空间定义和目录一致即可，例如：

我们实例化

```
$class = new \Think\Cache\Driver\File();
```

系统则会自动加载 `thinkphp/library/Think/Cache/Driver/File.php` 文件。

根命名空间（类库包）

根命名空间是一个关键的概念，以上面的 `\think\cache\driver\File` 类为例，`think` 就是一个根命名空间，其对应的初始命名空间目录就是系统的类库目录（`thinkphp/library/think`），我们可以简单的理解一个根命名空间对应了一个类库包。

| 系统内置的几个根命名空间（类库包）如下： | 名称 | 描述 | 类库目录 |
|----------------------|-----------|-------------------------|------|
| think | 系统核心类库 | thinkphp/library/think | |
| traits | 系统Trait类库 | thinkphp/library/traits | |
| app | 应用类库 | application | |

如果需要增加新的根命名空间，有两种方式：注册新的根命名空间或者放入 `EXTEND_PATH` 目录（自动注册）。

请注意本手册中的示例代码为了简洁，如无指定类库的命名空间的话，都表示指的是 `think` 命名空间，例如下面的代码：

```
Route::get('hello','index/hello');
```

请自行添加 `use think\Route` 或者使用

```
\think\Route::get('hello','index/hello');
```

自动注册

我们只需要把自己的类库包目录放入 `EXTEND_PATH` 目录（默认为 `extend`，可配置），就可以自动注册对应的命名空间，例如：

我们在 `extend` 目录下面新增一个 `my` 目录，然后定义一个 `\my\Test` 类（类文件位于 `extend/my/Test.php`）如下：

```
namespace my;

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

我们就可以直接实例化和调用：

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

如果我们在应用入口文件中重新定义了 `EXTEND_PATH` 常量的话，还可以改变 `\my\Test` 类文件的位置，例如：

```
define('EXTEND_PATH', '../vendor/');
```

那么 `\my\Test` 类文件的位置就变成了 `/vendor/my/File.php`。

手动注册

也可以通过手动注册的方式注册新的根命名空间，例如：

在应用入口文件中添加下面的代码：

```
\think\Loader::addNamespace('my', '../application/extend/my/');
```

如果要同时注册多个根命名空间，可以使用：

```
\think\Loader::addNamespace([
    'my' => '../application/extend/my/',
    'org' => '../application/extend/org/',
]);
```

也可以直接在应用的配置文件中添加配置，系统会在应用执行的时候自动注册。

```
'root_namespace' => [
    'my' => '../application/extend/my/',
    'org' => '../application/extend/org/',
]
```

应用类库包

为了避免和 `Composer` 自动加载的类库存在冲突，应用类库的命名空间的根都统一以 `app` 命名，例如：

```
namespace app\index\model;

class User extends \think\Model
{
}
```

其类文件位于 `application/index/model/User.php`。

```
namespace app\admin\event;

class User
{
}
```

其类文件位于 `application/admin/event/User.php` 。

如果觉得 `app` 根命名空间不合适或者有冲突，可以在应用配置文件中修改：

```
'app_namespace' => 'application',
```

定义后，应用类库的命名空间改为：

```
namespace application\index\model;

class User extends \think\Model
{
}
```

命名空间别名

框架允许给命名空间定义别名，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
}
```

原来在控制器里面调用方式为：

```
namespace app\index\controller;

use app\index\model\User;

class Index
{
    public function index()
    {
        $user = new User();
    }
}
```

如果我们在应用公共文件中注册命名空间别名如下：

```
\think\Loader::addNamespaceAlias('model', 'app\index\model');
```

那么，上面的控制器代码就可以更改为：

```
namespace app\index\controller;

use model\User;
```

```
class Index
{
    public function index()
    {
        $user = new User();
    }
}
```

本手册后续的章节，均建立在你已经了解PHP命名空间的基础之上，如果不掌握请自行补充PHP基础，否则你在后续的文档和ThinkPHP5.0的学习过程中，对命名空间的缺乏理解会成为你最大的学习障碍。

自动加载

概述

ThinkPHP5.0 真正实现了按需加载，所有类库采用自动加载机制，并且支持类库映射和 `composer` 类库的自动加载。

自动加载的实现由 `think\Loader` 类库完成，自动加载规范符合PHP的 `PSR-4`。

自动加载

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

类库的自动加载检测顺序如下：

- 1、类库映射检测；
- 2、`PSR-4` 自动加载检测；
- 3、`PSR-0` 自动加载检测；

系统会按顺序检测，一旦检测生效的话，就会自动载入对应的类库文件。

类库映射

遵循我们上面的命名空间定义规范的话，基本上可以完成类库的自动加载了，但是如果定义了较多的命名空间的话，效率会有所下降，所以，我们可以给常用的类库定义类库映射。命名类库映射相当于给类文件定义了一个别名，效率会比命名空间定位更高效，例如：

```
Loader::addClassMap('think\Log', LIB_PATH.'think\Log.php');
Loader::addClassMap('org\util\Array', LIB_PATH.'org\util\Array.php');
```

也可以利用 `addClassMap` 方法批量导入类库映射定义，例如：

```
$map = [
    'think\Log'      => LIB_PATH.'think\Log.php',
    'org\util\array' => LIB_PATH.'org\util\Array.php'
];
Loader::addClassMap($map);
```

虽然通过类库映射的方式注册的类可以不强制要求对应命名空间目录，但是仍然建议遵循PSR-4规范定义类库和目录。

类库导入

如果你不需要系统的自动加载功能，又或者没有使用命名空间的话，那么也可以使用 `think\Loader` 类的 `import` 方法手动加载类库文件，例如：

```
Loader::import('org.util.array');
Loader::import('@.util.upload');
```

示例

```
// 引入 extend/qrcode.php
Loader::import('qrcode', EXTEND_PATH);
// 助手函数
import('qrcode', EXTEND_PATH);

// 引入 extend/wechat-sdk/wechat.class.php
Loader::import('wechat-sdk.wechat', EXTEND_PATH, '.class.php');
// 助手函数
import('wechat-sdk.wechat', EXTEND_PATH, '.class.php');
```

| 类库导入也采用类似命名空间的概念（但不需要实际的命名空间支持），支持的“根命名空间”包括： | 目录 | 说明 |
|---|------------|----|
| behavior | 系统行为类库 | |
| think | 核心基类库 | |
| traits | 系统Traits类库 | |
| app | 应用类库 | |
| @ | 表示当前模块类库包 | |

如果完全遵从系统的命名空间定义的话，一般来说无需手动加载类库文件，直接实例化即可。

ThinkPHP5.0`不推荐使用`import`方法。

Composer自动加载

5.0版本支持`Composer`安装类库的自动加载，你可以直接按照`Composer`依赖库中的命名空间直接调用。

Traits引入

ThinkPHP 5.0 开始采用 `trait` 功能 (PHP5.4+) 来作为一种扩展机制，可以方便的实现一个类库的多继承问题。

`trait` 是一种为类似 PHP 的单继承语言而准备的代码复用机制。`trait` 为了减少单继承语言的限制，使开发人员能够自由地在不同层次结构内独立的类中复用方法集。`trait` 和类组合的语义是定义了一种方式来减少复杂性，避免传统多继承和混入类 (Mixin) 相关的典型问题。

但由于PHP5.4版本不支持 `trait` 的自动加载，因此如果是PHP5.4版本，必须手动导入 `trait` 类库，系统提供了一个助手函数 `load_trait`，用于自动加载 `trait` 类库，例如，可以这样正确引入 `trait` 类库。

```
namespace app\index\controller;

load_trait('controller/Jump'); // 引入traits\controller\Jump

class index
{
    use \traits\controller\Jump;

    public function index()
    {
        $this->assign('name', 'value');
        $this->show('index');
    }
}
```

如果你的PHP版本大于 5.5 的话，则可以省略 `load_trait` 函数引入 `trait`。

```
namespace app\index\controller;

class index
{
    use \traits\controller\Jump;

    public function index()
    {

    }
}
```

可以支持同时引入多个 `trait` 类库，例如：

```
namespace app\index\controller;

load_trait('controller/Other');
load_trait('controller/Jump');

class index
```

```
{
    use \traits\controller\Other;
    use \traits\controller\Jump;

    public function index()
    {
    }
}
```

或者使用

```
namespace app\index\controller;

load_trait('controller/Other');
load_trait('controller/Jump');

class index
{
    use \traits\controller\Other,\traits\controller\Jump;

    public function index()
    {
    }
}
```

系统提供了一些封装好的 `trait` 类库，主要是用于控制器和模型类的扩展。这些系统内置的 `trait` 类库的根命名空间采用 `traits` 而不是 `trait`，是为了避免和系统的关键字冲突。

`trait` 方式引入的类库需要注意优先级，从基类继承的成员将被 `trait` 插入的成员所覆盖。优先顺序是来自当前类的成员覆盖了 `trait` 的方法，而 `trait` 则覆盖了被继承的方法。

`trait` 类中不支持定义类的常量，在 `trait` 中定义的属性将不能在当前类中或者继承的类中重新定义。

冲突的解决

我们可以在一个类库中引入多个 `trait` 类库，如果两个 `trait` 都定义了一个同名的方法，如果没有明确解决冲突将会产生一个致命错误。

为了解决多个 `trait` 在同一个类中的命名冲突，需要使用 `insteadof` 操作符来明确指定使用冲突方法中的哪一个。

以上方式仅允许排除掉其它方法，`as` 操作符可以将其中一个冲突的方法以另一个名称来引入。

更多的关于 `trait` 内容可以参考[PHP官方手册](#)。

API友好

新版ThinkPHP针对 **API** 开发做了很多的优化，并且不依赖原来的API模式扩展。

数据输出

新版的控制器输出采用 **Response** 类统一处理，而不是直接在控制器中进行输出，通过设置 **default_return_type** 或者动态设置不同类型的 **Response** 输出就可以自动进行数据转换处理，一般来说，你只需要在控制器中返回字符串或者数组即可，例如如果我们配置：

```
'default_return_type'=>'json'
```

那么下面的控制器方法返回值会自动转换为json格式并返回。

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        $data = ['name'=>'thinkphp', 'url'=>'thinkphp.cn'];
        return ['data'=>$data, 'code'=>1, 'message'=>'操作完成'];
    }
}
```

访问该请求URL地址后，最终可以在浏览器中看到输出结果如下：

```
{"data":{"name":"thinkphp","url":"thinkphp.cn"},"code":1,"message":"\u64cd\u4f5c\u5b8c\u6210"}
```

如果你需要返回其他的数据格式的话，控制器本身的代码无需做任何改变。

支持明确指定输出类型的方式输出，例如下面指定 **JSON** 数据输出：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        $data = ['name'=>'thinkphp', 'url'=>'thinkphp.cn'];
        // 指定json数据输出
        return json(['data'=>$data, 'code'=>1, 'message'=>'操作完成']);
    }
}
```

或者指定输出 **XML** 类型数据：

本文档使用 [看云](#) 构建

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        $data = ['name'=>'thinkphp','url'=>'thinkphp.cn'];
        // 指定xml数据输出
        return xml(['data'=>$data,'code'=>1,'message'=>'操作完成']);
    }
}
```

核心支持的数据类型包括 `view`、`xml`、`json` 和 `jsonp`，其他类型的需要自己扩展。

错误调试

由于 API 开发不方便在客户端进行开发调试，但 ThinkPHP5 的 Trace 调试功能支持 Socket 在內的方式，可以实现远程的开发调试。

设置方式：

```
'app_trace' => true,
'trace'      => [
    'type'          => 'socket',
    // socket服务器
    'host'          => 'slog.thinkphp.cn',
],
```

然后安装 `chrome` 浏览器插件后即可进行远程调试，详细参考调试部分。

多层MVC

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

模型（Model）层

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

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

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

而这三个模型操作类统一都继承 `\think\Model` 类即可。

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

视图（View）层

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

```
view/default/user/add.html
view/blue/user/add.html
```

复杂一点的多层视图还可以更进一步，采用不同的视图目录来完成，例如：

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

控制器（Controller）层

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

```
controller/User //用于用户的业务逻辑控制和调度
event/User //用于用户的事件响应操作
```

`controller\User` 负责外部交互响应，通过URL请求响应，例如 `http://serverName/User/index`，而 `event\User` 负责内部的事件响应，并且只能在内部调用，所以是和外部隔离的。多层控制器的划分也不是强制的，可以根据应用的需要自由分层。控制器分层里面可以根据需要调用分层模型，也可以调用不同的分层视图。

配置

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

对于有些简单的应用，你无需配置任何配置文件，而对于复杂的要求，你还可以扩展自己的独立配置文件。

系统的配置参数是通过静态变量全局存取的，存取方式简单高效。

配置功能由 `\think\Config` 类完成。

配置格式

ThinkPHP支持多种格式的配置格式，但最终都是解析为PHP数组的方式。

PHP数组定义

返回**PHP数组**的方式是默认的配置定义格式，例如：

```
//项目配置文件
return [
    // 默认模块名
    'default_module'      => 'index',
    // 默认控制器名
    'default_controller'  => 'Index',
    // 默认操作名
    'default_action'      => 'index',
    //更多配置参数
    //...
];
```

配置参数名不区分大小写（因为无论大小写定义都会转换成小写），新版的建议是使用小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 [
    'cache'                => [
        'type'    => 'File',
        'path'    => CACHE_PATH,
        'prefix'  => '',
        'expire'  => 0,
    ],
];
```

其他配置格式支持

除了使用原生 PHP 数组之外，还可以使用 json/xml/ini 等其他格式支持（通过驱动的方式扩展）。

例如，我们可以使用下面的方式读取 json 配置文件：

```
Config::parse(APP_PATH.'config/config.json');
```

ini格式配置示例：

```
default_module=Index ;默认模块
default_controller=index ;默认控制器
default_action=index ;默认操作
```


xml格式配置示例：

```
<config>
<default_module>Index</default_module>
<default_controller>index</default_controller>
<default_action>index</default_action>
</config>
```

json格式配置示例：

```
{
  "default_module": "Index",
  "default_controller": "index",
  "default_action": "index"
}
```

二级配置

配置参数支持二级，例如，下面是一个二级配置的设置和读取示例：

```
$config = [
    'user' => [
        'type' => 1,
        'name' => 'thinkphp',
    ],
    'db' => [
        'type' => 'mysql',
        'user' => 'root',
        'password' => '',
    ],
];
// 设置配置参数
Config::set($config);
// 读取二级配置参数
echo Config::get('user.type');
// 或者使用助手函数
echo config('user.type');
```

系统不支持二级以上的配置参数读取，需要手动分步骤读取。

有作用域的情况下，仍然支持二级配置的操作。

如果采用其他格式的配置文件的话，二级配置定义方式如下（以ini和xml为例）：

```
[user]
type=1
name=thinkphp
[db]
type=mysql
user=rot
password=''
```

标准的xml格式文件定义：

```
<config>
<user>
<type>1</type>
<name>thinkphp</name>
</user>
<db>
<type>mysql</type>
<user>root</user>
<password></password>
</db>
</config>
```

set方法也支持二级配置，例如：

```
Config::set([
    'type'      => 'file',
    'prefix'    => 'think'
], 'cache');
```

配置加载

在ThinkPHP中，一般来说应用的配置文件是自动加载的，加载的顺序是：

惯例配置->应用配置->扩展配置->场景配置->模块配置->动态配置

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

下面说明下不同的配置文件的区别和位置：

惯例配置

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

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

应用配置

应用配置文件是应用初始化的时候首先加载的公共配置文件，默认位于 `application/config.php` 。

扩展配置

扩展配置文件是由 `extra_config_list` 配置参数定义的额外的配置文件，默认会加载 `database` 和 `validate` 两个扩展配置文件。

场景配置

每个应用都可以在不同的情况下设置自己的状态（或者称之为应用场景），并且加载不同的配置文件。

举个例子，你需要在公司和家里分别设置不同的数据库测试环境。那么可以这样处理，在公司环境中，我们在应用配置文件中配置：

```
'app_status'=>'office'
```

那么就会自动加载该状态对应的配置文件（默认位于 `application/office.php` ）。

如果我们回家后，我们修改定义为：

```
'app_status'=>'home'
```

那么就会自动加载该状态对应的配置文件（位于 `application/home.php` ）。

状态配置文件是可选的

模块配置

每个模块会自动加载自己的配置文件（位于 `application/当前模块名/config.php`）。

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

模块配置文件是可选的

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

加载配置文件

```
Config::load('配置文件名');
```

配置文件一般位于 `APP_PATH` 目录下面，如果需要加载其它位置的配置文件，需要使用完整路径，例如：

```
Config::load(APP_PATH.'config/config.php');
```

系统默认的配置定义格式是PHP返回数组的方式，例如：

```
return [
    '配置参数1'=>'配置值',
    '配置参数1'=>'配置值',
    // ... 更多配置
];
```

如果你定义格式是其他格式的话，可以使用 `parse` 方法来导入，例如：

```
Config::parse(APP_PATH.'my_config.ini', 'ini');
Config::parse(APP_PATH.'my_config.xml', 'xml');
```

`parse`方法的第一个参数需要传入完整的文件名或者配置内容。

如果不传入第二个参数的话，系统会根据配置文件名自动识别配置类型，所以下面的写法仍然是支持的：

```
Config::parse('my_config.ini');
```

`parse`方法除了支持读取配置文件外，也支持直接传入配置内容，例如：

```
$config = 'var1=val
var2=val';
Config::parse($config, 'ini');
```

支持传入配置文件内容的时候 第二个参数必须显式指定。

标准的ini格式文件定义：

```
配置参数1=配置值  
配置参数2=配置值
```

标准的xml格式文件定义：

```
<config>  
  <var1>val1</var1>  
  <var2>val2</var2>  
</config>
```

配置类采用驱动方式支持各种不同的配置文件类型，因此可以根据需要随意扩展。

更改配置目录

如果不希望配置文件放到应用目录下面，可以在入口文件中定义独立的配置目录，添加 `CONF_PATH` 常量定义即可，例如：

```
// 定义配置文件目录和应用目录同级  
define('CONF_PATH', '../config/');
```

配置目录下面的结构类似如下：

| | |
|-----------------------|---------------|
| ├application | 应用目录 |
| ├config | 配置目录 |
| │ ├──config.php | 应用配置文件 |
| │ ├──database.php | 数据库配置文件 |
| │ ├──route.php | 路由配置文件 |
| │ ├──... | 其它配置文件 |
| │ └index | index模块配置文件目录 |
| │ ├──config.php | index模块配置文件 |
| │ └... | index模块其它配置文件 |

读取配置

读取配置参数

设置完配置参数后，就可以使用get方法读取配置了，例如：

```
echo Config::get('配置参数1');
```

系统定义了一个助手函数 `config`，以上可以简化为：

```
echo config('配置参数1');
```

读取所有的配置参数：

```
dump(Config::get());  
// 或者 dump(config());
```

或者你需要判断是否存在某个设置参数：

```
Config::has('配置参数2');  
// 或者 config('?配置参数2');
```

如果需要读取二级配置，可以使用：

```
echo Config::get('配置参数.二级参数');  
echo config('配置参数.二级参数');
```

动态配置

设置配置参数

使用 `set` 方法动态设置参数，例如：

```
Config::set('配置参数','配置值');  
// 或者使用助手函数  
config('配置参数','配置值');
```

也可以批量设置，例如：

```
Config::set([  
    '配置参数1'=>'配置值',  
    '配置参数2'=>'配置值'  
]);  
// 或者使用助手函数  
config([  
    '配置参数1'=>'配置值',  
    '配置参数2'=>'配置值'  
]);
```

独立配置

独立配置文件

新版支持配置文件分离（也称为扩展配置），只需要配置 `extra_config_list` 参数(在应用公共配置文件中)。

例如，不使用独立配置文件的话，数据库配置信息应该是在 `config.php` 中配置如下：

```
/* 数据库设置 */
'database'          => [
    // 数据库类型
    'type'            => 'mysql',
    // 服务器地址
    'hostname'        => '127.0.0.1',
    // 数据库名
    'database'        => 'thinkphp',
    // 数据库用户名
    'username'        => 'root',
    // 数据库密码
    'password'        => '',
    // 数据库连接端口
    'hostport'        => '',
    // 数据库连接参数
    'params'          => [],
    // 数据库编码默认采用utf8
    'charset'         => 'utf8',
    // 数据库表前缀
    'prefix'          => '',
    // 数据库调试模式
    'debug'           => false,
],
```

如果需要使用独立配置文件的话，则首先在`config.php`中添加配置：

```
'extra_config_list' => ['database'],
```

定义之后，数据库配置就可以独立使用 `database.php` 文件，配置内容如下：

```
/* 数据库设置 */
return [
    // 数据库类型
    'type'            => 'mysql',
    // 服务器地址
    'hostname'        => '127.0.0.1',
    // 数据库名
    'database'        => 'thinkphp',
    // 数据库用户名
    'username'        => 'root',
    // 数据库密码
    'password'        => '',
    // 数据库连接端口
    'hostport'        => '',
```



```
// 数据库连接参数
'params'      => [],
// 数据库编码默认采用utf8
'charset'     => 'utf8',
// 数据库表前缀
'prefix'      => '',
// 数据库调试模式
'debug'       => false,
],
```

如果配置了 `extra_config_list` 参数，并同时在 `config.php` 和 `database.php` 文件中都配置的话，则 `database.php` 文件的配置会覆盖 `config.php` 中的设置。

独立配置文件的参数获取都是二维配置方式，例如，要获取 `database` 独立配置文件的 `type` 参数，应该是：

```
Config::get('database.type');
```

要获取完整的独立配置文件的参数，则使用：

```
Config::get('database');
```

系统默认设置了2个独立配置文件，包括 `database` 和 `validate`，分别用于设置数据库配置和验证规则定义。

配置作用域

作用域

配置参数支持作用域的概念，默认情况下，所有参数都在同一个系统默认作用域下面。如果你的配置参数需要用于不同的项目或者相互隔离，那么就可以使用作用域功能，作用域的作用好比是配置参数的命名空间一样。

```
// 导入my_config.php中的配置参数，并纳入user作用域
Config::load('my_config.php','', 'user');
// 解析并导入my_config.ini 中的配置参数，读入test作用域
Config::parse('my_config.ini', 'ini', 'test');
// 设置user_type参数，并纳入user作用域
Config::set('user_type', 1, 'user');
// 批量设置配置参数，并纳入test作用域
Config::set($config, 'test');
// 读取user作用域的user_type配置参数
echo Config::get('user_type', 'user');
// 读取user作用域下面的所有配置参数
dump(Config::get('', 'user'));
dump(Config::get('', null, 'user')); // 同上
// 判断在test作用域下面是否存在user_type参数
Config::has('user_type', 'test');
```

可以使用 `range` 方法切换当前配置文件的作用域，例如：

```
Config::range('test');
```

环境变量配置

环境变量配置

ThinkPHP5.0 支持使用环境变量配置。

在开发过程中，可以在应用根目录下面的 `.env` 来模拟环境变量配置，`.env` 文件中的配置参数定义格式采用 `ini` 方式，例如：

```
app_debug = true
app_trace = true
```

如果你的部署环境单独配置了环境变量，那么请删除 `.env` 配置文件，避免冲突。

环境变量配置的参数会全部转换为大写，值为 `null`，`no` 和 `false` 等效于 `""`，值为 `yes` 和 `true` 等效于 `"1"`。

ThinkPHP5.0默认的环境变量前缀是 `PHP_`，也可以通过改变 `ENV_PREFIX` 常量来重新设置。

注意，环境变量不支持数组参数，如果需要使用数组参数可以，使用下划线分割定义配置参数名：

```
database_username = root
database_password = 123456
```

或者使用

```
[database]
username = root
password = 123456
```

获取环境变量的值可以使用下面的两种方式获取：

```
Env::get('database.username');
Env::get('database.password');
// 同时下面的方式也可以获取
Env::get('database_username');
Env::get('database_password');
```

可以支持默认值，例如：

```
// 获取环境变量 如果不存在则使用默认值root
Env::get('database.username','root');
```

可以直接在应用配置中使用环境变量，例如：

```
return [  
  'hostname' => Env::get('hostname', '127.0.0.1'),  
];
```

环境变量中设置的 `app_debug` 和 `app_trace` 参数会自动生效（优先于应用的配置文件），其它参数则必须通过 `Env::get` 方法才能读取。

路由

路由功能由 `\think\Route` 类完成。

概述

由于 ThinkPHP5.0 默认采用的URL规则是：

<http://server/module/controller/action/param/value/...>

路由的作用是简化URL访问地址，并根据定义的路由类型做出正确的解析。

新版的路由功能做了大量的增强，包括：

- 支持路由到模块的控制器/操作、控制器类的方法、闭包函数和重定向地址，甚至是任何类库的方法；
- 闭包路由的增强；
- 规则路由支持全局和局部变量规则定义（正则）；
- 支持路由到任意层次的控制器；
- 子域名路由功能改进；
- 支持路由分组并支持分组参数定义；
- 增加资源路由和嵌套支持；
- 支持使用行为或者自定义函数检测路由规则；

ThinkPHP5.0的路由支持三种方式的URL解析规则。

5.0的路由是针对应用而不是针对模块，因此路由的设置也是针对应用下面的所有模块，如果希望不同的模块区分不同的设置（例如某些模块需要关闭路由，某些模块需要强制路由等），需要给该模块增加单独的入口文件，并作如下修改：

```
// 定义项目路径
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架基础文件
require __DIR__ . '/../thinkphp/base.php';
// 绑定当前入口文件到admin模块
\think\Route::bind('admin');
// 关闭admin模块的路由
\think\App::route(false);
// 执行应用
\think\App::run()->send();
```

路由模式

ThinkPHP5.0 的路由比较灵活，并且不需要强制定义，可以总结归纳为如下三种方式：

一、普通模式

关闭路由，完全使用默认的 `PATH_INFO` 方式URL：

```
'url_route_on' => false,
```

路由关闭后，不会解析任何路由规则，采用默认的 `PATH_INFO` 模式访问URL：

```
http://serverName/index.php/module/controller/action/param/value/...
```

但仍然可以通过操作方法的参数绑定、空控制器和空操作等特性实现URL地址的简化。

可以设置 `url_param_type` 配置参数来改变pathinfo模式下面的参数获取方式，默认是按名称成对解析，支持按照顺序解析变量，只需要更改为：

```
// 按照顺序解析变量  
'url_param_type' => 1,
```

二、混合模式

开启路由，并使用路由定义+默认 `PATH_INFO` 方式的混合：

```
'url_route_on' => true,  
'url_route_must'=> false,
```

该方式下面，只需要对需要定义路由规则的访问地址定义路由规则，其它的仍然按照第一种普通模式的 `PATH_INFO` 模式访问URL。

三、强制模式

开启路由，并设置必须定义路由才能访问：

```
'url_route_on' => true,  
'url_route_must'=> true,
```

这种方式下面必须严格给每一个访问地址定义路由规则（包括首页），否则将抛出异常。

首页的路由规则采用 `/` 定义即可，例如下面把网站首页路由输出 `Hello,world!`

```
Route::get('/',function(){  
    return 'Hello,world!';  
});
```

路由定义

注册路由规则

路由注册可以采用方法动态单个和批量注册，也可以直接定义路由定义文件的方式进行集中注册。

动态注册

路由定义采用 `\think\Route` 类的 `rule` 方法注册，通常是在应用的路由配置文件 `application/route.php` 进行注册，格式是：

```
Route::rule('路由表达式','路由地址','请求类型','路由参数（数组）','变量规则（数组）');
```

例如注册如下路由规则：

```
use think\Route;
// 注册路由到index模块的News控制器的read操作
Route::rule('new/:id','index/News/read');
```

我们访问：

```
http://serverName/new/5
```

ThinkPHP5.0的路由规则定义是从根目录开始，而不是基于模块名的。

会自动路由到：

```
http://serverName/index/news/read/id/5
```

并且原来的访问地址会自动失效。

路由表达式（第一个参数）支持定义命名标识，例如：

```
// 定义new路由命名标识
Route::rule(['new','new/:id'],'index/News/read');
```

注意，路由命名标识必须唯一，定义后可以用于URL的快速生成。

可以在`rule`方法中指定请求类型，不指定的话默认为任何请求类型，例如：

```
Route::rule('new/:id','News/update','POST');
```


表示定义的路由规则在POST请求下才有效。

| 请求类型包括： | 类型 | 描述 |
|---------|----------|----|
| GET | GET请求 | |
| POST | POST请求 | |
| PUT | PUT请求 | |
| DELETE | DELETE请求 | |
| * | 任何请求类型 | |

注意：请求类型参数必须大写。

系统提供了为不同的请求类型定义路由规则的简化方法，例如：

```
Route::get('new/:id','News/read'); // 定义GET请求路由规则
Route::post('new/:id','News/update'); // 定义POST请求路由规则
Route::put('new/:id','News/update'); // 定义PUT请求路由规则
Route::delete('new/:id','News/delete'); // 定义DELETE请求路由规则
Route::any('new/:id','News/read'); // 所有请求都支持的路由规则
```

如果要定义get和post请求支持的路由规则，也可以用：

```
Route::rule('new/:id','News/read','GET|POST');
```

我们也可以批量注册路由规则，例如：

```
Route::rule(['new/:id'=>'News/read','blog/:name'=>'Blog/detail']);
Route::get(['new/:id'=>'News/read','blog/:name'=>'Blog/detail']);
Route::post(['new/:id'=>'News/update','blog/:name'=>'Blog/detail']);
```

注册多个路由规则后，系统会依次遍历注册过的满足请求类型的路由规则，一旦匹配到正确的路由规则后则开始调用控制器的操作方法，后续规则就不再检测。

路由表达式

路由表达式统一使字符串定义，采用规则定义的方式。

正则路由定义功能已经废除，改由变量规则定义完成。

规则表达式

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

```
'/' => 'index', // 首页访问路由
'my' => 'Member/myinfo', // 静态地址路由
'blog/:id' => 'Blog/read', // 静态地址和动态地址结合
```

```
'new/:year/:month/:day'=>'News/read', // 静态地址和动态地址结合
':user/:blog_id'=>'Blog/read', // 全动态地址
```

规则表达式的定义以 `/` 为参数分割符（无论你的PATH_INFO分隔符设置是什么，请确保在定义路由规则表达式的时候统一使用 `/` 进行URL参数分割）。

每个参数中以`:`开头的参数都表示动态变量，并且会自动绑定到操作方法的对应参数。

可选定义

支持对路由参数的可选定义，例如：

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

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

以上定义路由规则后，下面的URL访问地址都可以被正确的路由匹配：

```
http://serverName/index.php/blog/2015
http://serverName/index.php/blog/2015/12
```

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

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

完全匹配

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

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

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

会匹配成功,而

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

则不会匹配成功。

如果是采用

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

方式定义的话，则两种方式的URL访问都可以匹配成功。

如果你希望所有的路由定义都是完全匹配的话，可以直接配置

```
// 开启路由定义的全局完全匹配
'route_complete_match' => true,
```

当开启全局完全匹配的时候，如果个别路由不需要使用完整匹配，可以添加路由参数覆盖定义：

```
Route::rule('new/:id', 'News/read', 'GET|POST', ['complete_match' => false]);
```

额外参数

在路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

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

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

批量注册

批量注册路由规则可以使用两种方式，包括方法注册和路由配置定义。

批量注册

如果不希望一个个注册，可以使用批量注册，规则如下：

```
Route::rule([
    '路由规则1'=>'路由地址和参数',
    '路由规则2'=>['路由地址和参数','匹配参数（数组）','变量规则（数组）'],
    ...
], '', '请求类型','匹配参数（数组）','变量规则');
```

如果在外面和规则里面同时传入了匹配参数和变量规则的话，路由规则定义里面的最终生效，但请求类型参数以最外层决定，例如：

```
Route::rule([
    'new/:id' => 'News/read',
    'blog/:id' => ['Blog/update',['ext'=>'shtml'],['id'=>'\d{4}']],
    ...
], '', 'GET',['ext'=>'html'],['id'=>'\d+']);
```

以上的路由注册，最终 `blog/:id` 只会在匹配shtml后缀的访问请求，id变量的规则则是 `\d{4}`。

如果不同的请求类型的路由规则是一样的，可以采用下面的方式定义：

```
Route::rule([
    ['blog/:id','Blog/read',['method'=>'get']],
    ['blog/:id','Blog/update',['method'=>'post']],
    ...
], '', '*', ['ext'=>'html'],['id'=>'\d+']);
```

上面的两条路由规则是相同，为了避免数组索引冲突的问题，可以改为如上方式进行注册。

同样，我们也可以使用其他几个注册方法进行批量注册。

```
// 批量注册GET路由
Route::get([
    'new/:id' => 'News/read',
    'blog/:id' => ['Blog/edit',[],['id'=>'\d+']]
    ...
]);
// 效果等同于
Route::rule([
    'new/:id' => 'News/read',
    'blog/:id' => ['Blog/edit',[],['id'=>'\d+']]
    ...
], '', 'GET');
```

定义路由配置文件

除了支持动态注册，也可以直接在应用目录下面的 `route.php` 的最后通过返回数组的方式直接定义路由规则，内容示例如下：

```
return [  
    'new/:id'    => 'News/read',  
    'blog/:id'   => ['Blog/update', ['method' => 'post|put'], ['id' => '\d+']],  
];
```

路由配置文件定义的路由规则效果和使用 `any` 注册路由规则一样。

路由动态注册和配置定义的方式可以共存，例如：

```
use think\Route;  
  
Route::rule('hello/:name', 'index/index/hello');  
  
return [  
    'new/:id'    => 'News/read',  
    'blog/:id'   => ['Blog/update', ['method' => 'post|put'], ['id' => '\d+']],  
];
```

默认情况下，只会加载一个路由配置文件 `route.php`，如果你需要定义多个路由文件，可以修改 `route_config_file` 配置参数，例如：

```
// 定义路由配置文件（数组）  
'route_config_file' => ['route', 'route1', 'route2'],
```

如果存在相同的路由规则，一样可以参考前面的批量注册方式进行定义。

由于检测机制问题，动态注册的性能比路由配置要高一些，尤其是多种请求类型混合定义的时候。

变量规则

变量规则

ThinkPHP5.0支持在规则路由中为变量用正则的方式指定变量规则，弥补了动态变量无法限制具体的类型问题，并且支持全局规则设置。使用方式如下：

全局变量规则

设置全局变量规则，全部路由有效：

```
// 设置name变量规则（采用正则定义）
Route::pattern('name', '\w+');
// 支持批量添加
Route::pattern([
    'name' => '\w+',
    'id'    => '\d+',
]);
```

局部变量规则

局部变量规则，仅在当前路由有效：

```
// 定义GET请求路由规则 并设置name变量规则
Route::get('new/:name', 'News/read', [], ['name'=>'\w+']);
```

如果一个变量同时定义了全局规则和局部规则，局部规则会覆盖全局变量的定义。

完整URL规则

如果要对整个URL进行规则检查，可以进行 `__url__` 变量规则，例如：

```
// 定义GET请求路由规则 并设置完整URL变量规则
Route::get('new/:id', 'News/read', [], ['__url__'=>'new\/\w+$']);
```

组合变量

如果你的路由规则比较特殊，可以在路由定义的时候使用组合变量。

例如：

```
Route::get('item-<name>-<id>', 'product/detail', [], ['name'=>'\\w+', 'id'=>'\\d+']);
```

组合变量的优势是路由规则中没有固定的分隔符，可以随意组合需要的变量规则，例如路由规则改成如下一样可以支持：

```
Route::get('item<name><id>', 'product/detail', [], ['name'=>'\\w+', 'id'=>'\\d+']);  
Route::get('item@<name>-<id>', 'product/detail', [], ['name'=>'\\w+', 'id'=>'\\d+']);
```

如果需要使用可选变量，则可以使用：

```
Route::get('item-<name><id?>', 'product/detail', [], ['name'=>'\\w+', 'id'=>'\\d+']);
```

路由参数

路由参数

路由参数是指可以设置一些路由匹配的条件参数，主要用于验证当前的路由规则是否有效，主要包括：

| 参数 | 说明 |
|------------------|--------------------|
| method | 请求类型检测，支持多个请求类型 |
| ext | URL后缀检测，支持匹配多个后缀 |
| deny_ext | URL禁止后缀检测，支持匹配多个后缀 |
| https | 检测是否https请求 |
| domain | 域名检测 |
| before_behavior | 前置行为（检测） |
| after_behavior | 后置行为（执行） |
| callback | 自定义检测方法 |
| merge_extra_vars | 合并额外参数 |

这些路由参数可以混合使用，只要有任何一条参数检查不通过，当前路由就不会生效，继续检测后面的路由规则。

请求类型

如果指定请求类型注册路由的话，无需设置 `method` 请求类型参数。如果使用了 `rule` 或者 `any` 方法注册路由，或者使用路由配置定义文件的话，可以单独使用 `method` 参数进行请求类型检测。

使用方法：

```
// 检测路由规则仅GET请求有效
Route::any('new/:id','News/read',['method'=>'get']);
// 检测路由规则仅GET和POST请求有效
Route::any('new/:id','News/read',['method'=>'get|post']);
```

URL后缀

```
// 定义GET请求路由规则 并设置URL后缀为html的时候有效
Route::get('new/:id','News/read',['ext'=>'html']);
```

支持匹配多个后缀，例如：

```
Route::get('new/:id','News/read',['ext'=>'shtml|html']);
```

可以设置禁止访问的URL后缀，例如：


```
// 定义GET请求路由规则 并设置禁止URL后缀为png、jpg和gif的访问
Route::get('new/:id','News/read',['deny_ext'=>'jpg|png|gif']);
```

域名检测

支持使用完整域名或者子域名进行检测，例如：

```
// 完整域名检测 只在news.thinkphp.cn访问时路由有效
Route::get('new/:id','News/read',['domain'=>'news.thinkphp.cn']);
// 子域名检测
Route::get('new/:id','News/read',['domain'=>'news']);
```

HTTPS检测

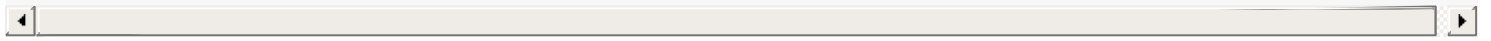
支持检测当前是否HTTPS访问

```
Route::get('new/:id','News/read',['https'=>true]);
```

前置行为检测

支持使用行为对路由进行检测是否匹配，如果行为方法返回false表示当前路由规则无效。

```
Route::get('new/:id','News/read',['before_behavior'=>'app\index\behavior\RouteCheck']);
```



行为类定义如下：

```
namespace app\index\behavior;

class RouteCheck
{
    public function run(&$url)
    {
        if('new/0'==$url){
            return false;
        }
    }
}
```

后置行为执行

可以为某个路由或者某个分组路由定义后置行为执行，表示当路由匹配成功后，执行的行为，例如：

```
Route::get('new/:id','News/read',['after_behavior'=>'app\index\behavior\ReadInit']);
```

其中\app\index\behavior\ReadInit 行为类定义如下：

```
namespace app\index\behavior;
class ReadInit {
    public function run(&$route){
        // 可以改变路由地址
    }
}
```

```

        $route .= '?status=1';
    }
}

```

如果成功匹配到 `new/:id` 路由后，就会执行行为类的run方法，参数是路由地址，可以动态改变。

Callback检测

也可以支持使用函数检测路由，如果函数返回false则表示当前路由规则无效，例如：

```
Route::get('new/:id', 'News/read', ['callback'=>'my_check_fun']);
```

合并额外参数

通常用于完整匹配的情况，如果有额外的参数则合并作为变量值，例如：

```
Route::get('new/:name$', 'News/read', ['merge_extra_vars'=>true]);
```

<http://serverName/new/thinkphp/hello>

会被匹配到，并且 `name` 变量的值为 `thinkphp/hello`。

配置文件中添加路由参数

如果使用配置文件的话，可以使用：

```

return [
    'blog/:id' => ['Blog/update', ['method' => 'post', 'ext'=>'html|shtml']],
];

```

路由地址

路由地址定义

路由地址表示定义的路由表达式最终需要路由到的地址以及一些需要的额外参数，支持下面5种方式定义：

| 定义方式 | 定义格式 |
|---------------|---------------------------------------|
| 方式1：路由到模块/控制器 | '[模块/控制器/操作]?额外参数1=值1&额外参数2=值2...' |
| 方式2：路由到重定向地址 | '外部地址'（默认301重定向）或者 ['外部地址','重定向代码'] |
| 方式3：路由到控制器的方法 | '@[模块/控制器/]操作' |
| 方式4：路由到类的方法 | '\完整的命名空间类::静态方法' 或者 '\完整的命名空间类@动态方法' |
| 方式5：路由到闭包函数 | 闭包函数定义（支持参数传入） |

其中方式5我们将会在下一个章节闭包支持中详细描述。

路由到模块/控制器/操作

这是最常用的一种路由方式，把满足条件的路由规则路由到相关的模块、控制器和操作，然后由App类调度执行相关的操作。

同时会进行模块的初始化操作（包括配置读取、公共文件载入、行为定义载入、语言包载入等等）。

路由地址的格式为：

[模块/控制器/]操作?参数1=值1&参数2=值2...

解析规则是从操作开始解析，然后解析控制器，最后解析模块，例如：

```
// 路由到默认或者绑定模块
'blog/:id'=>'blog/read',
// 路由到index模块
'blog/:id'=>'index/blog/read',
```

Blog类定义如下：

```
namespace app\index\controller;

class Blog {
    public function read($id){
        return 'read:'.$id;
    }
}
```

路由地址中支持多级控制器，使用下面的方式进行设置：

```
'blog/:id'=>'index/group.blog/read'
```

表示路由到下面的控制器类，

```
index/controller/group/Blog
```

Blog类定义如下：

```
namespace app\index\controller\group;

class Blog {
    public function read($id){
        return 'read:'.$id;
    }
}
```

还可以支持路由到动态的模块、控制器或者操作，例如：

```
// action变量的值作为操作方法传入
':action/blog/:id' => 'index/blog/:action'
// 变量传入index模块的控制器和操作方法
':c/:a'=> 'index/:c/:a'
```

如果关闭路由功能的话，默认也会按照该规则对URL进行解析调度。

额外参数

在这种方式路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

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

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

路由到操作方法

路由地址的格式为：

@[模块/控制器/]操作

这种方式看起来似乎和第一种是一样的，本质的区别是直接执行某个控制器类的方法，而不需要去解析 **模块/控制器/操作** 这些，同时也不会去初始化模块。

例如，定义如下路由后：

```
'blog/:id'=>'@index/blog/read',
```

系统会直接执行

```
Loader::action('index/blog/read');
```

相当于直接调用 `\app\index\controller\blog` 类的 `read` 方法。

Blog 类定义如下：

```
namespace app\index\controller;

class Blog {
    public function read($id){
        return 'read:'.$id;
    }
}
```

通常这种方式下面，由于没有定义当前模块名、当前控制器名和当前方法名，从而导致视图的默认模板规则失效，所以这种情况下面，如果使用了视图模板渲染，则必须传入明确的参数。

路由到类的方法

路由地址的格式为（动态方法）：

`\类的命名空间\类名@方法名`

或者（静态方法）

`\类的命名空间\类名::方法名`

这种方式更进一步，可以支持执行任何类的方法，而不仅仅是执行控制器的操作方法，例如：

```
'blog/:id'=>\app\index\service\Blog@read',
```

执行的是 `\app\index\service\Blog` 类的 `read` 方法。

也支持执行某个静态方法，例如：

```
'blog/:id'=>\app\index\service\Blog::read',
```

路由到重定向地址

重定向的外部地址必须以 `"/` 或者 `http` 开头的地址。

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

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

和

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

虽然都是路由到同一个地址，但是前者采用的是301重定向的方式路由跳转，这种方式的好处是URL可以比较随意（包括可以在URL里面传入更多的非标准格式的参数），而后者只是支持模块和操作地址。举个例子，如果我们希望 **avatar/123** 重定向到 `/member/avatar/id/123_small`的话，只能使用：

```
'avatar/:id'=>'/member/avatar/id/:id_small'
```

路由地址采用重定向地址的话，如果要引用动态变量，直接使用动态变量即可。

采用重定向到外部地址通常对网站改版后的URL迁移过程非常有用，例如：

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

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

资源路由

资源路由

5.0支持设置 RESTFu1 请求的资源路由，方式如下：

```
Route::resource('blog','index/blog');
```

或者在路由配置文件中 使用 `__rest__` 添加资源路由定义：

```
return [
    // 定义资源路由
    '__rest__'=>[
        // 指向index模块的blog控制器
        'blog'=>'index/blog',
    ],
    // 定义普通路由
    'hello/:id'=>'index/hello',
]
```

设置后会自动注册7个路由规则，如下：

| 标识 | 请求类型 | 生成路由规则 | 对应操作方法（默认） |
|--------|--------|---------------|------------|
| index | GET | blog | index |
| create | GET | blog/create | create |
| save | POST | blog | save |
| read | GET | blog/:id | read |
| edit | GET | blog/:id/edit | edit |
| update | PUT | blog/:id | update |
| delete | DELETE | blog/:id | delete |

具体指向的控制器由路由地址决定，例如上面的设置，会对应index模块的blog控制器，你只需要为Blog控制器创建以上对应的操作方法就可以支持下面的URL访问：

```
http://serverName/blog/
http://serverName/blog/128
http://serverName/blog/28/edit
```

Blog控制器中的对应方法如下：

```
namespace app\index\controller;
class Blog {
    public function index(){
    }
```

```
public function read($id){
}

public function edit($id){
}
}
```

可以改变默认id参数名，例如：

```
Route::resource('blog', 'index/blog', ['var'=>['blog'=>'blog_id']]);
```

控制器的方法定义需要调整如下：

```
namespace app\index\controller;
class Blog {
    public function index(){
    }

    public function read($blog_id){
    }

    public function edit($blog_id){
    }
}
```

也可以在定义资源路由的时候限定执行的方法（标识），例如：

```
// 只允许index read edit update 四个操作
Route::resource('blog', 'index/blog', ['only'=>['index', 'read', 'edit', 'update']]);
// 排除index和delete操作
Route::resource('blog', 'index/blog', ['except'=>['index', 'delete']]);
```

资源路由的标识不可更改，但生成的路由规则和对对应操作方法可以修改。

如果需要更改某个资源路由标识的对应操作，可以使用下面方法：

```
Route::rest('create', ['GET', '/add', 'add']);
```

设置之后，URL访问变为：

```
http://serverName/blog/create
变成
http://serverName/blog/add
```

创建blog页面的对应的操作方法也变成了add。

支持批量更改，如下：


```
Route::rest([
    'save' => ['POST', '', 'store'],
    'update' => ['PUT', '/:id', 'save'],
    'delete' => ['DELETE', '/:id', 'destory'],
]);
```

资源嵌套

支持资源路由的嵌套，例如：

```
Route::resource('blog.comment', 'index/comment');
```

就可以访问如下地址：

```
http://serverName/blog/128/comment/32
http://serverName/blog/128/comment/32/edit
```

生成的路由规则分别是：

```
blog/:blog_id/comment/:id
blog/:blog_id/comment/:id/edit
```

Comment控制器对应的操作方法如下：

```
namespace app\index\controller;
class Comment{
    public function edit($id,$blog_id){
    }
}
```

edit方法中的参数顺序可以随意，但参数名称必须满足定义要求。

如果需要改变其中的变量名，可以使用：

```
// 更改嵌套资源路由的blog资源的资源变量名为blogId
Route::resource('blog.comment', 'index/comment', ['var'=>['blog'=>'blogId']]);
```

Comment控制器对应的操作方法改变为：

```
namespace app\index\controller;

class Comment{
    public function edit($id,$blogId)
    {
    }
}
```


快捷路由

快捷路由允许你快速给控制器注册路由，并且针对不同的请求类型可以设置方法前缀，例如：

```
// 给User控制器设置快捷路由
Route::controller('user', 'index/User');
```

User控制器定义如下：

```
namespace app\index\controller;

class User {
    public function getInfo()
    {
    }

    public function getPhone()
    {
    }

    public function postInfo()
    {
    }

    public function putInfo()
    {
    }

    public function deleteInfo()
    {
    }
}
```

我们可以通过下面的URL访问

```
get http://localhost/user/info
get http://localhost/user/phone
post http://localhost/user/info
put http://localhost/user/info
delete http://localhost/user/info
```

路由别名

路由别名功能可以使用一条规则，批量定义一系列的路由规则。

例如，我们希望使用 `user` 可以访问index模块的User控制器的所有操作，可以使用：

```
// user 别名路由到 index/User 控制器
Route::alias('user', 'index/User');
```

如果在路由配置文件 `route.php` 中定义的话，使用：

```
return [
    '__alias__' => [
        'user' => 'index/User',
    ],
];
```

和前面的方式是等效的。

然后可以直接通过URL地址访问User控制器的操作，例如：

```
http://serverName/index.php/user/add
http://serverName/index.php/user/edit/id/5
http://serverName/index.php/user/read/id/5
```

如果URL参数绑定方式使用按顺序绑定的话，URL地址可以进一步简化，参考请求->方法参数绑定。

路由别名可以指向任意一个有效的路由地址，例如下面指向一个类

```
// user 路由别名指向 User控制器类
Route::alias('user', '\app\index\controller\User');
```

路由别名不支持变量类型和路由条件判断，单纯只是为了缩短URL地址，并且在定义的时候需要注意避免和路由规则产生混淆。

支持给路由别名设置路由条件，例如：

```
// user 别名路由到 index/user 控制器
Route::alias('user', ['index/user', ['ext'=>'html']]);
```

或者在路由配置文件中使用：

```
return [  
  '__alias__' => [  
    'user' => ['index/user', ['ext'=>'html']],  
  ],  
];
```

路由分组

路由分组

路由分组功能允许把相同前缀的路由定义合并分组，这样可以提高路由匹配的效率，不必每次都去遍历完整的路由规则。

例如，我们有定义如下两个路由规则的话

```
'blog/:id'    => ['Blog/read', ['method' => 'get'], ['id' => '\d+']],
'blog/:name' => ['Blog/read', ['method' => 'post']],
```

可以合并到一个blog分组

```
'[blog]'      => [
  ':id'    => ['Blog/read', ['method' => 'get'], ['id' => '\d+']],
  ':name' => ['Blog/read', ['method' => 'post']],
],
```

可以使用 `Route` 类的 `group` 方法进行注册，如下：

```
Route::group('blog', [
  ':id'    => ['Blog/read', ['method' => 'get'], ['id' => '\d+']],
  ':name' => ['Blog/read', ['method' => 'post']],
]);
```

可以给分组路由定义一些公用的路由设置参数，例如：

```
Route::group('blog', [
  ':id'    => ['Blog/read', [], ['id' => '\d+']],
  ':name' => ['Blog/read', [],
], ['method'=>'get', 'ext'=>'html']);
```

支持使用闭包方式注册路由分组，例如：

```
Route::group('blog', function(){
  Route::rule(':id', 'blog/read', [], ['id'=>'\d+']);
  Route::rule(':name', 'blog/read', [], ['name'=>'\w+']);
}, ['method'=>'get', 'ext'=>'html']);
```

如果仅仅是用于对一些路由规则设置一些公共的路由参数，也可以使用：

```
Route::group(['method'=>'get', 'ext'=>'html'], function(){
  Route::rule('blog/:id', 'blog/read', [], ['id'=>'\d+']);
  Route::rule('blog/:name', 'blog/read', [], ['name'=>'\w+']);
});
```

路由分组支持嵌套，例如：

```
Route::group(['method'=>'get', 'ext'=>'html'], function(){
    Route::group('blog', function(){
        Route::rule('blog/:id', 'blog/read', [], ['id'=>'\\d+']);
        Route::rule('blog/:name', 'blog/read', [], ['name'=>'\\w+']);
    });
});
```

MISS路由

全局MISS路由

如果希望在没有匹配到所有的路由规则后执行一条设定的路由，可以使用 **MISS** 路由功能，只需要在路由配置文件中定义：

```
return [
  'new/:id'    => 'News/read',
  'blog/:id'   => ['Blog/update', ['method' => 'post|put'], ['id' => '\d+']],
  '__miss__'   => 'public/miss',
];
```

或者使用 `miss` 方法注册路由

```
Route::miss('public/miss');
```

当没有匹配到所有的路由规则后，会路由到 `public/miss` 路由地址。

分组MISS路由

分组支持独立的 **MISS** 路由，例如如下定义：

```
return [
  'blog' => [
    'edit/:id' => ['Blog/edit', ['method' => 'get'], ['id' => '\d+']],
    ':id'      => ['Blog/read', ['method' => 'get'], ['id' => '\d+']],
    '__miss__' => 'blog/miss',
  ],
  'new/:id'   => 'News/read',
  '__miss__'  => 'public/miss',
];
```

如果使用 `group` 方法注册路由的话，可以使用下面的方式：

```
Route::group('blog', function(){
  Route::rule(':id', 'blog/read', [], ['id'=>'\d+']);
  Route::rule(':name', 'blog/read', [], ['name'=>'\w+']);
  Route::miss('blog/miss');
}, ['method'=>'get', 'ext'=>'html']);
```


闭包支持

闭包定义

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

```
Route::get('hello',function(){  
    return 'hello,world!';  
});
```

参数传递

闭包定义的时候支持参数传递，例如：

```
Route::get('hello/:name',function($name){  
    return 'Hello, '.$name;  
});
```

规则路由中定义的动态变量的名称 就是闭包函数中的参数名称，不分次序。

因此，如果我们访问的URL地址是：

```
http://serverName/hello/thinkphp
```

则浏览器输出的结果是：

```
Hello, thinkphp
```

路由绑定

可以使用路由绑定简化URL或者路由规则的定义，绑定支持如下方式：

绑定到模块/控制器/操作

把当前的URL绑定到模块/控制器/操作，最多支持绑定到操作级别，例如在路由配置文件中添加：

```
// 绑定当前的URL到 index模块
Route::bind('index');
// 绑定当前的URL到 index模块的blog控制器
Route::bind('index/blog');
// 绑定当前的URL到 index模块的blog控制器的read操作
Route::bind('index/blog/read');
```

该方式针对路由到模块/控制器/操作有效，假如我们绑定到了index模块的blog控制器，那么原来的访问URL从

```
http://serverName/index/blog/read/id/5
```

可以简化成

```
http://serverName/read/id/5
```

如果定义了路由

```
Route::get('index/blog/:id', 'index/blog/read');
```

那么访问URL就变成了

```
http://serverName/5
```

绑定到命名空间

把当前的URL绑定到某个指定的命名空间，例如：

```
// 绑定命名空间
Route::bind('\app\index\controller', 'namespace');
```

那么，我们接下来只需要通过

```
http://serverName/blog/read/id/5
```

就可以直接访问 `\app\index\controller\Blog` 类的read方法。

绑定到类

把当前的URL直接绑定到某个指定的类，例如：

```
// 绑定到类
Route::bind('\app\index\controller\Blog', 'class');
```

那么，我们接下来只需要通过

```
http://serverName/read/id/5
```

就可以直接访问 `\app\index\controller\Blog` 类的read方法。

注意：绑定到命名空间和类之后，不会进行模块的初始化工作。

入口文件绑定

如果我们需要给某个入口文件绑定模块，可以使用下面两种方式：

常量定义

只需要入口文件添加 `BIND_MODULE` 常量，即可把当前入口文件绑定到指定的模块或者控制器，例如：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 绑定到index模块
define('BIND_MODULE', 'index');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
```

自动入口绑定

如果你的入口文件都是对应实际的模块名，那么可以使用入口文件自动绑定模块的功能，只需要在应用配置文件中添加：

```
// 开启入口文件自动绑定模块
'auto_bind_module' => true,
```

当我们重新添加一个 `public/demo.php` 入口文件，内容和 `public/index.php` 一样：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
```

但其实访问 `demo.php` 的时候，其实已经自动绑定到了 `demo` 模块。

绑定模型

路由规则和分组支持绑定模型数据，例如：

```
Route::rule('hello/:id','index/index/hello','GET',[
    'ext' => 'html',
    'bind_model' => [
        'user' => '\app\index\model\User',
    ],
]);
```

会自动给当前路由绑定 id为 当前路由变量值的User模型数据。

可以定义模型数据的查询条件，例如：

```
Route::rule('hello/:name/:id','index/index/hello','GET',[
    'ext' => 'html',
    'bind_model' => [
        'user' => ['\app\index\model\User','id&name']
    ],
]);
```

也可以使用闭包来返回模型对象数据

```
Route::rule('hello/:id','index/index/hello','GET',[
    'ext' => 'html',
    'bind_model' => [
        'user' => function($route){
            $model = new \app\index\model\User;
            return $model->where($route)->find();
        }
    ],
]);
```

闭包函数的参数就是当前请求的URL变量信息。

在控制器中可以通过下面的代码获取：

```
request()->user;
```

域名路由

ThinkPHP支持完整域名、子域名和IP部署的路由和绑定功能，同时还可以起到简化URL的作用。

要启用域名部署路由功能，首先需要开启：

```
'url_domain_deploy' => true
```

定义域名部署规则支持两种方式：动态注册和配置定义。

动态注册

可以在应用的公共文件或者配置文件中动态注册域名部署规则，例如：

```
// blog子域名绑定到blog模块
Route::domain('blog', 'blog');
// 完整域名绑定到admin模块
Route::domain('admin.thinkphp.cn', 'admin');
// IP绑定到admin模块
Route::domain('114.23.4.5', 'admin');
```

blog子域名绑定后，URL访问规则变成：

```
// 原来的URL访问
http://www.thinkphp.cn/blog/article/read/id/5
// 绑定到blog子域名访问
http://blog.thinkphp.cn/article/read/id/5
```

支持绑定的时候添加默认参数，例如：

```
// blog子域名绑定到blog模块
Route::domain('blog', 'blog?var=thinkphp');
```

除了绑定到模块之外，还隐式传入了一个 `$_GET['var'] = 'thinkphp'` 变量。

支持直接绑定到控制器，例如：

```
// blog子域名绑定到index模块的blog控制器
Route::domain('blog', 'index/blog');
```

URL访问地址变化为：

```
// 原来的URL访问
http://www.thinkphp.cn/index/blog/read/id/5
// 绑定到blog子域名访问
```

```
http://blog.thinkphp.cn/read/id/5
```

如果你的域名后缀比较特殊，例如是 `com.cn` 或者 `net.cn` 之类的域名，需要配置：

```
'url_domain_root'=>'thinkphp.com.cn'
```

泛域名部署

可以支持泛域名部署规则，例如：

```
// 绑定泛二级域名域名到book模块
Route::domain('*', 'book?name=*');
```

下面的URL访问都会直接访问book模块

```
http://hello.thinkphp.cn
http://quickstart.thinkphp.cn
```

并且可以直接通过`$_GET['name']`变量 获取当前的泛域名。

支持三级泛域名部署，例如：

```
// 绑定泛三级域名到user模块
Route::domain('*.user', 'user?name=*');
```

如果我们访问如下URL地址：

```
http://hello.user.thinkphp.cn
```

的同时，除了会访问user模块之外，还会默认传入 `$_GET['name'] = 'hello'`

在配置传入参数的时候，如果需要使用当前的泛域名作为参数，可以直接设置为`"*"`即可。

目前只支持二级域名和三级域名的泛域名部署。

配置定义方式

除了动态注册之外，还支持直接在路由配置文件中定义域名部署规则，例如：

```
return [
    '__domain__'=>[
        'blog'      => 'blog',
        // 泛域名规则建议在最后定义
        '*.user'     => 'user',
        '*'          => 'book',
    ],
];
```

```
// 下面是路由规则定义
]
```

域名绑定地址

前面我们看到的域名部署规则：

```
// blog子域名绑定到blog模块
Route::domain('blog', 'blog');
```

其实是把域名绑定到模块的方式，其实还有其他的绑定方式。

绑定到命名空间

```
// blog子域名绑定命名空间
Route::domain('blog', '\app\blog\controller');
```

绑定到类

```
// blog子域名绑定到类
Route::domain('blog', '@app\blog\controller\Article');
```

绑定到闭包函数

如果需要，你也可以直接把域名绑定到一个闭包函数，例如：

```
// blog子域名绑定闭包函数
Route::domain('blog', function(){
    echo 'hello';
    return ['bind'=>'module', 'module'=>'blog'];
});
```

域名绑定到闭包函数其实是一种劫持，可以在闭包函数里面动态注册其它的绑定机制或者注册新的路由，例如：

```
Route::domain('www', function(){
    // 动态注册域名的路由规则
    Route::rule('new/:id', 'index/news/read');
    Route::rule(':user', 'index/user/info');
});
```

如果你不希望继续，可以直接在闭包函数里面中止执行。

```
// blog子域名绑定到闭包函数
Route::domain('blog', function(){
    exit('hello');
});
```

绑定路由规则

可以把域名绑定到一系列指定的路由规则，例如：

```
Route::domain('blog',[
    // 动态注册域名的路由规则
    ':id' => ['blog/read', ['method'=>'GET'], ['id'=>'\d+']],
    ':name'=>'blog/read',
]);
```

如果使用配置文件配置的话，可以按照下面的方式：

```
return [
    '__domain__'=>[
        'blog' => [
            // 动态注册域名的路由规则
            ':id' => ['blog/read', ['method'=>'GET'], ['id'=>'\d+']],
            ':name'=>'blog/read',
        ],
    ],
    // 下面是其它的路由规则定义
];
```

更详细的绑定功能请参考后面的路由绑定一章内容。

URL生成

ThinkPHP5.0支持路由URL地址的统一生成，并且支持所有的路由方式，以及完美解决了路由地址的反转解析，无需再为路由定义和变化而改变URL生成。

URL生成使用 `\think\Url::build()` 方法或者使用系统提供的助手函数 `url()`，参数一致：

```
Url::build('地址表达式',['参数'],['URL后缀'],['域名'])
```

```
url('地址表达式',['参数'],['URL后缀'],['域名'])
```

地址表达式和参数

对使用不同的路由地址方式，地址表达式的定义有所区别。参数单独通过第二个参数传入，假设我们定义了一个路由规则如下：

```
Route::rule('blog/:id', 'index/blog/read');
```

就可以使用下面的方式来生成URL地址：

```
Url::build('index/blog/read', 'id=5&name=thinkphp');  
Url::build('index/blog/read', ['id'=>5, 'name'=>'thinkphp']);  
url('index/blog/read', 'id=5&name=thinkphp');  
url('index/blog/read', ['id'=>5, 'name'=>'thinkphp']);
```

下面我们统一使用第一种方式讲解。

使用模块/控制器/操作生成

如果你的路由方式是路由到模块/控制器/操作，那么可以直接写

```
// 生成index模块 blog控制器的read操作 URL访问地址  
Url::build('index/blog/read', 'id=5&name=thinkphp');  
// 使用助手函数  
url('index/blog/read', 'id=5&name=thinkphp');
```

以上方法都会生成下面的URL地址：

```
/index.php/blog/5/name/thinkphp.html
```

注意，生成方法的第一个参数必须和路由定义的路由地址保持一致，如果写成下面的方式可能无法正确生成URL地址：

```
Url::build('blog/read', 'id=5&name=thinkphp');
```

如果你的环境支持REWRITE，那么生成的URL地址会变为：

```
/blog/5/name/thinkphp.html
```

如果你配置了：

```
'url_common_param'=>true
```

那么生成的URL地址变为：

```
/index.php/blog/5.html?name=thinkphp
```

不在路由规则里面的变量会直接使用普通URL参数的方式。

需要注意的是，URL地址生成不会检测路由的有效性，只是按照给定的路由地址和参数生成符合条件的路由规则。

使用控制器的方法生成

如果你的路由地址是采用控制器的方法，并且路由定义如下：

```
// 这里采用配置方式定义路由 动态注册的方式一样有效
'blog/:id' => '@index/blog/read'
```

那么可以使用如下方式生成：

```
// 生成index模块 blog控制器的read操作 URL访问地址
Url::build('@index/blog/read', 'id=5');
// 使用助手函数
url('@index/blog/read', 'id=5');
```

那么自动生成的URL地址变为：

```
/index.php/blog/5.html
```

使用类的方法生成

如果你的路由地址是路由到类的方法，并且做了如下路由规则定义：

```
// 这里采用配置方式定义路由 动态注册的方式一样有效
Route::rule(['blog', 'blog/:id'], '\app\index\controller\blog@read');
```

如果路由地址是到类的方法，需要首先给路由定义命名标识，然后使用标识快速生成URL地址。

那么可以使用如下方式生成：

```
// 生成index模块 blog控制器的read操作 URL访问地址
Url::build('blog?id=5');
url('blog?id=5');
```

那么自动生成的URL地址变为：

```
/index.php/blog/5.html
```

直接使用路由地址

我们也可以直接使用路由地址来生成URL，例如：

我们定义了路由规则如下：

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

可以使用下面的方式直接使用路由规则生成URL地址：

```
Url::build('/blog/5');
```

那么自动生成的URL地址变为：

```
/index.php/blog/5.html
```

URL后缀

默认情况下，系统会自动读取 `url_html_suffix` 配置参数作为URL后缀（默认为html），如果我们设置了：

```
'url_html_suffix' => 'shtml'
```

那么自动生成的URL地址变为：

```
/index.php/blog/5.shtml
```

如果我们设置了多个URL后缀支持

```
'url_html_suffix' => 'html|shtml'
```

则会取第一个后缀来生成URL地址，所以自动生成的URL地址还是：

```
/index.php/blog/5.html
```

如果你希望指定URL后缀生成，则可以使用：

```
Url::build('index/blog/read','id=5','shtml');  
url('index/blog/read','id=5','shtml');
```

域名生成

默认生成的URL地址是不带域名的，如果你采用了多域名部署或者希望生成带有域名的URL地址的话，就需要传入第四个参数，该参数有两种用法：

自动生成域名

```
Url::build('index/blog/read','id=5','shtml',true);  
url('index/blog/read','id=5','shtml',true);
```

第四个参数传入 `true` 的话，表示自动生成域名，如果你开启了 `url_domain_deploy` 还会自动识别匹配当前URL规则的域名。

例如，我们注册了域名路由信息如下：

```
Route::domain('blog','index/blog');
```

那么上面的URL地址生成为：

```
http://blog.thinkphp.cn/read/id/5.shtml
```

指定域名

你也可以显式传入需要生成地址的域名，例如：

```
Url::build('index/blog/read','id=5','shtml','blog');  
url('index/blog/read','id=5','shtml','blog');
```

或者传入完整的域名

```
Url::build('index/blog/read','id=5','shtml','blog.thinkphp.cn');  
U('index/blog/read','id=5','shtml','blog.thinkphp.cn');
```

生成的URL地址为：

```
http://blog.thinkphp.cn/read/id/5.shtml
```

也可以直接在第一个参数里面传入域名，例如：

```
Url::build('index/blog/read@blog','id=5');  
url('index/blog/read@blog','id=5');  
url('index/blog/read@blog.thinkphp.cn','id=5');
```

生成锚点

支持生成URL的锚点，可以直接在URL地址参数中使用：

```
Url::build('index/blog/read#anchor@blog','id=5');  
url('index/blog/read#anchor@blog','id=5');
```

锚点和域名一起使用的时候，注意锚点在前面，域名在后面。

生成的URL地址为：

```
http://blog.thinkphp.cn/read/id/5.html#anchor
```

隐藏或者加上入口文件

有时候我们生成的URL地址可能需要加上 `index.php` 或者去掉 `index.php`，大多数时候系统会自动判断，如果发现自动生成的地址有问题，可以直接在调用 `build` 方法之前调用 `root` 方法，例如加上 `index.php`：

```
Url::root('/index.php');  
Url::build('index/blog/read','id=5');
```

或者隐藏 `index.php`：

```
Url::root('/');  
Url::build('index/blog/read','id=5');
```

`root` 方法只需要调用一次即可。

控制器

[控制器定义](#)

[控制器初始化](#)

[前置操作](#)

[跳转和重定向](#)

[空操作](#)

[空控制器](#)

[多级控制器](#)

[分层控制器](#)

[Rest控制器](#)

[自动定位控制器](#)

[资源控制器](#)

控制器定义

ThinkPHP V5.0的控制器定义比较灵活，可以无需继承任何的基础类，也可以继承官方封装的 `\think\Controller` 类或者其他控制器类。

控制器定义

一个典型的控制器类定义如下：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'index';
    }
}
```

控制器类文件的实际位置是

```
application\index\controller\Index.php
```

控制器类可以无需继承任何类，命名空间默认以 `app` 为根命名空间。

控制器的根命名空间可以设置，例如我们在应用配置文件中修改：

```
// 修改应用类库命名空间
'app_namespace' => 'application',
```

则实际的控制器类应该更改定义如下：

```
namespace application\index\controller;

class Index
{
    public function index()
    {
        return 'index';
    }
}
```

只是命名空间改变了，但实际的文件位置和文件名并没有改变。

使用该方式定义的控制器类，如果要在控制器里面渲染模板，可以使用


```
namespace app\index\controller;

use think\View;

class Index
{
    public function index()
    {
        $view = new View();
        return $view->fetch('index');
    }
}
```

或者直接使用view助手函数渲染模板输出，例如：

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return view('index');
    }
}
```

如果继承了 `\think\Controller` 类的话，可以直接调用 `\think\View` 及 `\think\Request` 类的方法，例如：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 获取包含域名的完整URL地址
        $this->assign('domain',$this->request->url(true));
        return $this->fetch('index');
    }
}
```

渲染输出

默认情况下，控制器的输出全部采用 `return` 的方式，无需进行任何的手动输出，系统会自动完成渲染内容的输出。

下面都是有效的输出方式：

```
namespace app\index\controller;

class Index
{
    public function hello()
```

```

    {
        return 'hello,world!';
    }

    public function json()
    {
        return json_encode($data);
    }

    public function read()
    {
        return view();
    }
}

```

控制器一般不需要任何输出，直接return即可。

输出转换

默认情况下，控制器的返回输出不会做任何的数据处理，但可以设置输出格式，并进行自动的数据转换处理，前提是控制器的输出数据必须采用 **return** 的方式返回。

如果控制器定义为：

```

namespace app\index\controller;

class Index
{
    public function hello()
    {
        return 'hello,world!';
    }

    public function data()
    {
        return ['name'=>'thinkphp','status'=>1];
    }
}

```

当我们设置输出数据格式为JSON：

```

// 默认输出类型
'default_return_type' => 'json',

```

我们访问

```

http://localhost/index.php/index/Index/hello
http://localhost/index.php/index/Index/data

```

输出的结果变成：

```
"hello,world!"  
{"name":"thinkphp","status":1}
```

默认情况下，控制器在ajax请求会对返回类型自动转换，默认为json

如果我们控制器定义

```
namespace app\index\controller;  
  
class Index  
{  
    public function data()  
    {  
        return ['name'=>'thinkphp','status'=>1];  
    }  
}
```

我们访问

```
http://localhost/index.php/index/Index/data
```

输出的结果变成：

```
{"name":"thinkphp","status":1}
```

当我们设置输出数据格式为html：

```
// 默认输出类型  
'default_ajax_return' => 'html',
```

这种情况下ajax请求不会对返回内容进行转换

控制器初始化

如果你的控制器类继承了 `\think\Controller` 类的话，可以定义控制器初始化方法 `_initialize`，在该控制器的方法调用之前首先执行。

例如：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function _initialize()
    {
        echo 'init<br/>';
    }

    public function hello()
    {
        return 'hello';
    }

    public function data()
    {
        return 'data';
    }
}
```

如果访问

<http://localhost/index.php/index/Index/hello>

会输出

```
init
hello
```

如果访问

<http://localhost/index.php/index/Index/data>

会输出

```
init
data
```

前置操作

可以为某个或者某些操作指定前置执行的操作方法，设置 `beforeActionList` 属性可以指定某个方法为其他方法的前置操作，数组键名为需要调用的前置方法名，无值的话为当前控制器下所有方法的前置方法。

```
['except' => '方法名,方法名']
```

表示这些方法不使用前置方法，

```
['only' => '方法名,方法名']
```

表示只有这些方法使用前置方法。

示例如下:

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    protected $beforeActionList = [
        'first',
        'second' => ['except'=>'hello'],
        'three'  => ['only'=>'hello,data'],
    ];

    protected function first()
    {
        echo 'first<br/>';
    }

    protected function second()
    {
        echo 'second<br/>';
    }

    protected function three()
    {
        echo 'three<br/>';
    }

    public function hello()
    {
        return 'hello';
    }

    public function data()
    {
        return 'data';
    }
}
```

前置操作

访问

```
http://localhost/index.php/index/Index/hello
```

最后的输出结果是

```
first  
three  
hello
```

访问

```
http://localhost/index.php/index/Index/data
```

的输出结果是：

```
first  
second  
three  
data
```

跳转和重定向

页面跳转

在应用开发中，经常会遇到一些带有提示信息的跳转页面，例如操作成功或者操作错误页面，并且自动跳转到另外一个目标页面。系统的 `\think\Controller` 类内置了两个跳转方法 `success` 和 `error`，用于页面跳转提示。

使用方法很简单，举例如下：

```
namespace app\index\controller;

use think\Controller;
use app\index\model\User;

class Index extends Controller
{
    public function index()
    {
        $User = new User; //实例化User对象
        $result = $User->save($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');
        }
    }
}
```

跳转地址是可选的，`success`方法的默认跳转地址是 `$_SERVER["HTTP_REFERER"]`，`error`方法的默认跳转地址是 `javascript:history.back(-1);`。

默认的等待时间都是3秒

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

```
THINK_PATH . 'tpl/dispatch_jump.tpl'
```

我们可以改变默认的模板：

```
//默认错误跳转对应的模板文件
'dispatch_error.tpl' => APP_PATH . 'tpl/dispatch_jump.tpl',
//默认成功跳转对应的模板文件
'dispatch_success.tpl' => APP_PATH . 'tpl/dispatch_jump.tpl',
```

也可以使用项目内部的模板文件

```
//默认错误跳转对应的模板文件
'dispatch_error_tmpl' => 'public/error',
//默认成功跳转对应的模板文件
'dispatch_success_tmpl' => 'public/success',
```

模板文件可以使用模板标签，并且可以使用下面的模板变量：

| 变量 | 含义 |
|--------|-------------|
| \$data | 要返回的数据 |
| \$msg | 页面提示信息 |
| \$code | 返回的code |
| \$wait | 跳转等待时间 单位为秒 |
| \$url | 跳转页面地址 |

`error`方法会自动判断当前请求是否属于 `Ajax` 请求，如果属于 `Ajax` 请求则会自动转换为 `default_ajax_return` 配置的格式返回信息。`success`在 `Ajax` 请求下不返回信息，需要开发者自行处理。

重定向

`\think\Controller` 类的 `redirect` 方法可以实现页面的重定向功能。

`redirect`方法的参数用法和 `Url::build` 方法的用法一致（参考[URL生成部分](#)），例如：

```
//重定向到News模块的Category操作
$this->redirect('News/category', ['cate_id' => 2]);
```

上面的用法是跳转到News模块的category操作，重定向后会改变当前的URL地址。

或者直接重定向到一个指定的外部URL地址，例如：

```
//重定向到指定的URL地址 并且使用302
$this->redirect('http://thinkphp.cn/blog/2', 302);
```


空操作

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

例如，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City控制器类定义一个 `_empty`（空操作）方法：

```
<?php
namespace app\index\controller;

class City
{
    public function _empty($name)
    {
        //把所有城市的操作解析到city方法
        return $this->showCity($name);
    }

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

接下来，我们就可以在浏览器里面输入

```
http://serverName/index/city/beijing/
http://serverName/index/city/shanghai/
http://serverName/index/city/shenzhen/
```

由于City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
```

空控制器

空控制器的概念是指当系统找不到指定的控制器名称的时候，系统会尝试定位空控制器(Error)，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/city/shanghai/
```

变成

```
http://serverName/index/shanghai/
```

这样更加简单的方式，如果按照传统的模式，我们必须给每个城市定义一个控制器类，然后在每个控制器类的index方法里面进行处理。可是如果使用空控制器功能，这个问题就可以迎刃而解了。

我们可以给项目定义一个Error控制器类

```
<?php
namespace app\index\controller;

use think\Request;

class Error
{
    public function index(Request $request)
    {
        //根据当前控制器名来判断要执行那个城市的操作
        $cityName = $request->controller();
        return $this->city($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
        return '当前城市' . $name;
    }
}
```

接下来，我们就可以在浏览器里面输入

```
http://serverName/index/beijing/
http://serverName/index/shanghai/
http://serverName/index/shenzhen/
```

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

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

空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

空控制器Error是可以定义的

```
// 更改默认的空控制器名  
'empty_controller' => 'MyError',
```

当找不到控制器的时候，就会定位到MyError控制器类进行操作。

多级控制器

多级控制器

新版支持任意层次级别的控制器，并且支持路由，例如：

```
namespace app\index\controller\one;

use think\Controller;

class Blog extends Controller
{
    public function index()
    {
        return $this->fetch();
    }

    public function add()
    {
        return $this->fetch();
    }

    public function edit($id)
    {
        return $this->fetch();
    }
}
```

该控制器类的文件位置为：

```
application/index/controller/one/Blog.php
```

访问地址可以使用

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

如果要在路由定义中使用多级控制器，可以使用：

```
\think\Route::get('blog/add', 'index/one.blog/add');
```

分层控制器

访问控制器

ThinkPHP引入了分层控制器的概念，通过URL访问的控制器为访问控制器层（Controller）或者主控制器，访问控制器是由 `\think\App` 类负责调用和实例化的，无需手动实例化。

URL解析和路由后，会把当前的URL地址解析到【**模块/控制器/操作**】，其实也就是执行某个控制器类的某个操作方法，下面是一个示例：

```
namespace app\index\controller;

class Blog
{
    public function index()
    {
        return 'index';
    }

    public function add()
    {
        return 'add';
    }

    public function edit($id)
    {
        return 'edit:'.$id;
    }
}
```

当前定义的主控制器位于index模块下面，所以当访问不同的URL地址的页面输出如下：

```
http://serverName/index/blog/index // 输出 index
http://serverName/index/blog/add     // 输出 add
http://serverName/index/blog/edit/id/5 // 输出 edit:5
```

新版的控制器可以不需要继承任何基类，当然，你可以定义一个公共的控制器基础类来被继承，也可以通过控制器扩展来完成不同的功能（例如Restful实现）。

如果不经路由访问的话，URL中的控制器名会首先强制转为小写，然后再解析为驼峰法实例化该控制器。

分层控制器

除了访问控制器外，我们还可以定义其他分层控制器类，这些分层控制器是不能够被URL访问直接调用到的，只能在访问控制器、模型类的内部，或者视图模板文件中进行调用。

例如，我们定义Blog事件控制器如下：

```
namespace app\index\event;

class Blog
{
    public function insert()
    {
        return 'insert';
    }

    public function update($id)
    {
        return 'update:'.$id;
    }

    public function delete($id)
    {
        return 'delete:'.$id;
    }
}
```

定义完成后，就可以用下面的方式实例化并调用方法了：

```
$event = \think\Loader::controller('Blog', 'event');
echo $event->update(5); // 输出 update:5
echo $event->delete(5); // 输出 delete:5
```

为了方便调用，系统提供了controller助手函数直接实例化多层控制器，例如：

```
$event = controller('Blog', 'event');
echo $event->update(5); // 输出 update:5
echo $event->delete(5); // 输出 delete:5
```

支持跨模块调用，例如：

```
$event = controller('Admin/Blog', 'event');
echo $event->update(5); // 输出 update:5
```

表示实例化Admin模块的Blog控制器类，并执行update方法。

除了实例化分层控制器外，还可以直接调用分层控制器类的某个方法，例如：

```
echo \think\Loader::action('Blog/update', ['id' => 5], 'event'); // 输出 update:5
```

也可以使用助手函数 `action` 实现相同的功能：

```
echo action('Blog/update', ['id' => 5], 'event'); // 输出 update:5
```

利用分层控制器的机制，我们可以用来实现 **widget**（其实就是在模板中调用分层控制器），例如：

定义 `index\widget\Blog` 控制器类如下：

```
namespace app\index\widget;

class Blog {
    public function header()
    {
        return 'header';
    }

    public function left()
    {
        return 'left';
    }

    public function menu($name)
    {
        return 'menu:'.$name;
    }
}
```

我们在模板文件中就可以直接调用 `app\index\widget\Blog` 分层控制器了，使用助手函数 `action`

```
{:action('Blog/header', '', 'widget')}
{:action('Blog/menu', ['name' => 'think'], 'widget')}
```

框架还提供了 `widget` 函数用于简化 `Widget` 控制器的调用，可以直接使用助手函数 `widget`

```
{:widget('Blog/header')}
{:widget('Blog/menu', ['name' => 'think'])}
```

Rest控制器

Rest控制器

如果需要让你的控制器支持RESTful的话，可以使用Rest控制器，在定义访问控制器的时候直接继承 `think\controller\Rest` 即可，例如：

```
namespace app\index\controller;  
  
use think\controller\Rest;  
  
class Blog extends Rest  
{  
}
```

配合示例需要，我们首先在应用配置文件中添加：

```
// URL伪静态后缀  
'url_html_suffix'          => 'html|xml|json|jsonp',
```

RESTful方法定义

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，需要自己在操作方法里面添加判断代码，示例：

```
<?php  
  
namespace app\index\controller;  
  
use think\controller\Rest;  
  
class Blog extends Rest  
{  
    public function rest()  
    {  
        switch ($this->method){  
            case 'get': // get请求处理代码  
                if ($this->type == 'html'){  
                } elseif ($this->type == 'xml'){  
                }  
                break;  
            case 'put': // put请求处理代码  
                break;  
            case 'post': // post请求处理代码  
                break;  
        }  
    }  
}
```

在Rest操作方法中，可以使用 `$this->type` 获取当前访问的资源类型，用 `$this->method` 获取当前的请

求类型。

RESTFul 输出

使用Rest类提供的 response 方法

```
$this->response($data, 'json', 200);
```

使用 think\Response 类

```
Response::create($data, 'json')->code(200);
```

使用助手函数

```
json($data, 200);
```

\$data为需要输出的数据，第二个参数为输出数据的http状态码
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,jsonp,html等编码格式输出，例如：

```
// 输出 json 格式数据
json($data, 200);
// 输出 jsonp 格式数据
jsonp($data, 200);
// 输出 xml 格式数据
xml($data, 200);
```

| 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 restRequestMethod相同的时候，例如read_pdf | |
| 操作名_提交类型 | 当前资源后缀和 restDefaultType相同的时候，例如read_post | |

这种方式的rest方法定义采用了空操作机制，所以要使用这种方式的前提就是不能为当前操作定义方法，如果检测到相关的restful方法则不再检查后面的方法规范，例如我们定义了InfoController如下：

```
namespace app\index\controller;

use think\controller\Rest;

class Info extends Rest
{
    public function read_get_xml($id)
    {
```

```
        // 输出id为1的Info的XML数据
    }

    public function read_xml($id)
    {
        // 输出id为1的Info的XML数据
    }

    public function read_json($id)
    {
        // 输出id为1的Info的json数据
    }
}
```

如果我们访问的URL是：

```
http://serverName/index/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Info类的 read_get_xml方法，而不是read_xml方法，但是如果访问的URL是：

```
http://serverName/index/info/read/id/1.json
```

那么则会调用read_json方法。

自动定位控制器

如果你使用了多级控制器的话，可以设置 `controller_auto_search` 参数开启自动定位控制器，便于URL访问，例如首先在配置文件中添加：

```
'controller_auto_search' => true,
```

然后定义控制器如下：

```
namespace app\index\controller\one;

use think\Controller;

class Blog extends Controller
{
    public function index()
    {
        return $this->fetch();
    }

    public function add()
    {
        return $this->fetch();
    }

    public function edit($id)
    {
        return $this->fetch('edit:'.$id);
    }
}
```

我们就可以直接访问下面的URL地址了：

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

资源控制器

资源控制器可以让你轻松的创建 **RESTFu1** 资源控制器，可以通过命令行生成需要的资源控制器，例如：

```
// 生成index模块的Blog资源控制器
php think make:controller index/Blog
```

或者使用完整的命名空间生成

```
php think make:controller app\index\controller\Blog
```

然后你只需要为资源控制器注册一个资源路由：

```
Route::resource('blog','index/Blog');
```

设置后会自动注册7个路由规则，如下：

| 请求类型 | 生成路由规则 | 对应操作方法 |
|--------|----------------------|--------|
| GET | blog | index |
| GET | blog/create | create |
| POST | blog | save |
| GET | blog/:id | read |
| GET | blog/:id/edit | edit |
| PUT | blog/:id | update |
| DELETE | blog/:id | delete |

请求

[请求信息](#)

[输入变量](#)

[更改变量](#)

[请求类型](#)

[请求伪装](#)

[方法参数绑定](#)

[注入请求对象](#)

[HTTP头信息](#)

[方法注入](#)

[属性注入](#)

[伪静态](#)

请求信息

如果要获取当前的请求信息，可以使用 `\think\Request` 类，除了下文中的

```
$request = Request::instance();
```

也可以使用助手函数

```
$request = request();
```

当然，最方便的还是使用注入请求对象的方式来获取变量。

例如：

获取URL信息

```
$request = Request::instance();  
// 获取当前域名  
echo 'domain: ' . $request->domain() . '<br/>';  
// 获取当前入口文件  
echo 'file: ' . $request->baseFile() . '<br/>';  
// 获取当前URL地址 不含域名  
echo 'url: ' . $request->url() . '<br/>';  
// 获取包含域名的完整URL地址  
echo 'url with domain: ' . $request->url(true) . '<br/>';  
// 获取当前URL地址 不含QUERY_STRING  
echo 'url without query: ' . $request->baseUrl() . '<br/>';  
// 获取URL访问的ROOT地址  
echo 'root: ' . $request->root() . '<br/>';  
// 获取URL访问的ROOT地址  
echo 'root with domain: ' . $request->root(true) . '<br/>';  
// 获取URL地址中的PATH_INFO信息  
echo 'pathinfo: ' . $request->pathinfo() . '<br/>';  
// 获取URL地址中的PATH_INFO信息 不含后缀  
echo 'pathinfo: ' . $request->path() . '<br/>';  
// 获取URL地址中的后缀信息  
echo 'ext: ' . $request->ext() . '<br/>';
```

输出结果为：

```
domain: http://tp5.com  
file: /index.php  
url: /index/index/hello.html?name=thinkphp  
url with domain: http://tp5.com/index/index/hello.html?name=thinkphp  
url without query: /index/index/hello.html  
root:  
root with domain: http://tp5.com  
pathinfo: index/index/hello.html
```

```
pathinfo: index/index/hello
ext: html
```

设置/获取 模块/控制器/操作名称

```
$request = Request::instance();
echo "当前模块名称是" . $request->module();
echo "当前控制器名称是" . $request->controller();
echo "当前操作名称是" . $request->action();
```

如果当前访问的地址是 http://serverName/index.php/index/hello_world/index

输出结果为：

```
当前模块名称是index
当前控制器名称是HelloWorld
当前操作名称是index
```

设置模块名称值需要像module方法中传入名称即可，同样使用于设置控制器名称和操作名称

```
Request::instance()->module('module_name');
```

获取请求参数

```
$request = Request::instance();
echo '请求方法：' . $request->method() . '<br/>';
echo '资源类型：' . $request->type() . '<br/>';
echo '访问地址：' . $request->ip() . '<br/>';
echo '是否Ajax请求：' . var_export($request->isAjax(), true) . '<br/>';
echo '请求参数：';
dump($request->param());
echo '请求参数：仅包含name';
dump($request->only(['name']));
echo '请求参数：排除name';
dump($request->except(['name']));
```

输出结果为：

```
请求方法：GET
资源类型：html
访问地址：127.0.0.1
是否Ajax请求：false
请求参数：
array (size=2)
    'test' => string 'ddd' (length=3)
    'name' => string 'thinkphp' (length=8)

请求参数：仅包含name
array (size=1)
    'name' => string 'thinkphp' (length=8)

请求参数：排除name
array (size=1)
```

```
'test' => string 'ddd' (length=3)
```

获取路由和调度信息

hello方法修改如下：

```
$request = Request::instance();  
echo '路由信息：';  
dump($request->route());  
echo '调度信息：';  
dump($request->dispatch());
```

路由定义为：

```
return [  
    'hello/:name' =>['index/hello',[],['name'=>'\w+']],  
];
```

访问下面的URL地址：

```
http://serverName/hello/thinkphp
```

输出信息为：

```
路由信息：  
array (size=4)  
    'rule' => string 'hello/:name' (length=11)  
    'route' => string 'index/hello' (length=11)  
    'pattern' =>  
        array (size=1)  
            'name' => string '\w+' (length=3)  
    'option' =>  
        array (size=0)  
            empty  
  
调度信息：  
array (size=2)  
    'type' => string 'module' (length=6)  
    'module' =>  
        array (size=3)  
            0 => null  
            1 => string 'index' (length=5)  
            2 => string 'hello' (length=5)
```

设置请求信息

如果某些环境下面获取的请求信息有误，可以手动设置这些信息参数，使用下面的方式：

```
$request = Request::instance();  
$request->root('index.php');  
$request->pathinfo('index/index/hello');
```


输入变量

概述

可以通过 Request 对象完成全局输入变量的检测、获取和安全过滤，支持包括 `$_GET`、`$_POST`、`$_REQUEST`、`$_SERVER`、`$_SESSION`、`$_COOKIE`、`$_ENV` 等系统变量，以及文件上传信息。

检测变量是否设置

可以使用 has 方法来检测一个变量参数是否设置，如下：

```
Request::instance()->has('id','get');
Request::instance()->has('name','post');
```

或者使用助手函数

```
input('?get.id');
input('?post.name');
```

变量检测可以支持所有支持的系统变量。

变量获取

变量获取使用 `\think\Request` 类的如下方法及参数：

变量类型方法('变量名/变量修饰符','默认值','过滤方法')

变量类型方法包括：

| 方法 | 描述 |
|---------|-------------------------------|
| param | 获取当前请求的变量 |
| get | 获取 <code>\$_GET</code> 变量 |
| post | 获取 <code>\$_POST</code> 变量 |
| put | 获取 <code>PUT</code> 变量 |
| delete | 获取 <code>DELETE</code> 变量 |
| session | 获取 <code>\$_SESSION</code> 变量 |
| cookie | 获取 <code>\$_COOKIE</code> 变量 |
| request | 获取 <code>\$_REQUEST</code> 变量 |
| server | 获取 <code>\$_SERVER</code> 变量 |
| env | 获取 <code>\$_ENV</code> 变量 |
| route | 获取 路由（包括PATHINFO）变量 |
| file | 获取 <code>\$_FILES</code> 变量 |

获取 PARAM 变量

PARAM变量是框架提供的用于自动识别 GET 、 POST 或者 PUT 请求的一种变量获取方式，是系统推荐的获取请求参数的方法，用法如下：

```
// 获取当前请求的name变量
Request::instance()->param('name');
// 获取当前请求的所有变量（经过过滤）
Request::instance()->param();
// 获取当前请求的所有变量（原始数据）
Request::instance()->param(false);
// 获取当前请求的所有变量（包含上传文件）
Request::instance()->param(true);
```

param方法会把当前请求类型的参数和PATH_INFO变量以及GET请求合并。

使用助手函数实现：

```
input('param.name');
input('param.');
```

或者

```
input('name');
input('');
```

因为 input 函数默认就采用PARAM变量读取方式。

获取 GET 变量

```
Request::instance()->get('id'); // 获取某个get变量
Request::instance()->get('name'); // 获取get变量
Request::instance()->get(); // 获取所有的get变量（经过过滤的数组）
Request::instance()->get(false); // 获取所有的get变量（原始数组）
```

或者使用内置的助手函数 input 方法实现相同的功能：

```
input('get.id');
input('get.name');
input('get.');
```

注：pathinfo地址参数不能通过get方法获取，查看“获取PARAM变量”

获取 POST 变量

```
Request::instance()->post('name'); // 获取某个post变量
Request::instance()->post(); // 获取经过过滤的全部post变量
Request::instance()->post(false); // 获取全部的post原始变量
```

使用助手函数实现：

```
input('post.name');  
input('post.');
```

获取 PUT 变量

```
Request::instance()->put('name'); // 获取某个put变量  
Request::instance()->put(); // 获取全部的put变量（经过过滤）  
Request::instance()->put(false); // 获取全部的put原始变量
```

使用助手函数实现：

```
input('put.name');  
input('put.');
```

获取 REQUEST 变量

```
Request::instance()->request('id'); // 获取某个request变量  
Request::instance()->request(); // 获取全部的request变量（经过过滤）  
Request::instance()->request(); // 获取全部的request原始变量数据
```

使用助手函数实现：

```
input('request.id');  
input('request.');
```

获取 SERVER 变量

```
Request::instance()->server('PHP_SELF'); // 获取某个server变量  
Request::instance()->server(); // 获取全部的server变量
```

使用助手函数实现：

```
input('server.PHP_SELF');  
input('server.');
```

获取 SESSION 变量

```
Request::instance()->session('user_id'); // 获取某个session变量  
Request::instance()->session(); // 获取全部的session变量
```

使用助手函数实现：

```
input('session.user_id');  
input('session.');
```

获取 Cookie 变量

```
Request::instance()->cookie('user_id'); // 获取某个cookie变量
Request::instance()->cookie(); // 获取全部的cookie变量
```

使用助手函数实现：

```
input('cookie.user_id');
input('cookie.');
```

变量过滤

支持对获取的变量进行过滤，过滤方式包括函数、方法过滤，以及PHP内置的Types of filters，我们可以设置全局变量过滤方法，例如：

```
Request::instance()->filter('htmlspecialchars');
```

支持设置多个过滤方法，例如：

```
Request::instance()->filter(['strip_tags','htmlspecialchars'],
```

也可以在获取变量的时候添加过滤方法，例如：

```
Request::instance()->get('name','', 'htmlspecialchars'); // 获取get变量 并用htmlspecialchars函数过滤
Request::instance()->param('username','', 'strip_tags'); // 获取param变量 并用strip_tags函数过滤
Request::instance()->post('name','', 'org\Filter::safeHtml'); // 获取post变量 并用org\Filter类的safeHtml方法过滤
```

可以支持传入多个过滤规则，例如：

```
Request::instance()->param('username','', 'strip_tags, strtolower'); // 获取param变量 并依次调用strip_tags、 strtolower函数过滤
```

Request对象还支持PHP内置提供的Filter ID过滤，例如：

```
Request::instance()->post('email','', FILTER_VALIDATE_EMAIL);
```

框架对FilterID做了转换支持，因此也可以使用字符串的方式，例如：

```
Request::instance()->post('email','', 'email');
```

采用字符串方式定义 `FilterID` 的时候，系统会自动进行一次 `filter_id` 调用转换成 `Filter` 常量。

具体的字符串根据 `filter_list` 函数的返回值来定义。

需要注意的是，采用Filter ID 进行过滤的话，如果不符合过滤要求的话 会返回false，因此你需要配合默认值来确保最终的值符合你的规范。

例如，

```
Request::instance()->post('email','',FILTER_VALIDATE_EMAIL);
```

就表示，如果不是规范的email地址的话 返回空字符串。

如果希望和全局的过滤方法合并的话，可以使用

```
// 获取get变量 并使用全局函数htmlspecialchars函数以及strtolower方法过滤
Request::instance()->get('name','', 'strtolower',true);
```

获取部分变量

如果你只需要获取当前请求的部分参数，可以使用：

```
// 只获取当前请求的id和name变量
Request::instance()->only('id,name');
```

或者使用数组方式

```
// 只获取当前请求的id和name变量
Request::instance()->only(['id','name']);
```

默认获取的是当前请求参数，如果需要获取其它类型的参数，可以使用第二个参数，例如：

```
// 只获取GET请求的id和name变量
Request::instance()->only(['id','name'],'get');
// 只获取POST请求的id和name变量
Request::instance()->only(['id','name'],'post');
```

排除部分变量

也支持排除某些变量获取，例如

```
// 排除id和name变量
Request::instance()->except('id,name');
```

或者使用数组方式

```
// 排除id和name变量
Request::instance()->except(['id', 'name']);
```

同样支持指定变量类型获取：

```
// 排除GET请求的id和name变量
Request::instance()->except(['id', 'name'], 'get');
// 排除POST请求的id和name变量
Request::instance()->except(['id', 'name'], 'post');
```

变量修饰符

`input` 函数支持对变量使用修饰符功能，可以更好的过滤变量。

用法如下：

```
input('变量类型.变量名/修饰符');
```

或者

```
Request::instance()->变量类型('变量名/修饰符');
```

例如：

```
input('get.id/d');
input('post.name/s');
input('post.ids/a');
Request::instance()->get('id/d');
```

ThinkPHP5.0版本默认的变量修饰符是 `/s`，如果需要传入字符串之外的变量可以使用下面的修饰符，包括：

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

如果你要获取的数据为数组，请一定要注意要加上 `/a` 修饰符才能正确获取到。

更改变量

如果需要更改请求变量的值，可以通过下面的方式：

```
// 更改GET变量
Request::instance()->get(['id'=>10]);
// 更改POST变量
Request::instance()->post(['name'=>'thinkphp']);
```

尽量避免直接修改 `$_GET` 或者 `$_POST` 数据，同时也不能直接修改 `param` 变量，例如下面的操作是无效的：

```
// 更改请求变量
Request::instance()->param(['id'=>10]);
```


请求类型

获取请求类型

在很多情况下面，我们需要判断当前操作的请求类型是 `GET`、`POST`、`PUT`、`DELETE` 或者 `HEAD`，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。

ThinkPHP5.0 取消了用于判断请求类型的系统常量(如 `IS_GET`，`IS_POST`等)，统一采用 `think\Request` 类处理请求类型。

用法如下

```
// 是否为 GET 请求
if (Request::instance()->isGet()) echo "当前为 GET 请求";
// 是否为 POST 请求
if (Request::instance()->isPost()) echo "当前为 POST 请求";
// 是否为 PUT 请求
if (Request::instance()->isPut()) echo "当前为 PUT 请求";
// 是否为 DELETE 请求
if (Request::instance()->isDelete()) echo "当前为 DELETE 请求";
// 是否为 Ajax 请求
if (Request::instance()->isAjax()) echo "当前为 Ajax 请求";
// 是否为 Pjax 请求
if (Request::instance()->isPjax()) echo "当前为 Pjax 请求";
// 是否为手机访问
if (Request::instance()->isMobile()) echo "当前为手机访问";
// 是否为 HEAD 请求
if (Request::instance()->isHead()) echo "当前为 HEAD 请求";
// 是否为 Patch 请求
if (Request::instance()->isPatch()) echo "当前为 PATCH 请求";
// 是否为 OPTIONS 请求
if (Request::instance()->isOptions()) echo "当前为 OPTIONS 请求";
// 是否为 cli
if (Request::instance()->isCli()) echo "当前为 cli";
// 是否为 cgi
if (Request::instance()->isCgi()) echo "当前为 cgi";
```

助手函数

```
// 是否为 GET 请求
if (request()->isGet()) echo "当前为 GET 请求";
.....
```

通过注入请求对象的功能，可以更简单的实现。请参考控制器章节的注入请求对象的内容。

请求伪装

支持请求类型伪装，可以在 `POST` 表单里面提交 `_method` 变量，传入需要伪装的请求类型，例如：

```
<form method="post" action="">
  <input type="text" name="name" value="Hello">
  <input type="hidden" name="_method" value="PUT" >
  <input type="submit" value="提交">
</form>
```

提交后的请求类型会被系统识别为 `PUT` 请求。

你可以设置为任何合法的请求类型，包括 `GET`、`POST`、`PUT` 和 `DELETE` 等。

如果你需要改变伪装请求的变量名，可以修改应用配置文件：

```
// 表单请求类型伪装变量
'var_method'          => '_m',
```

方法参数绑定

方法参数绑定是把URL地址（或者路由地址）中的变量作为操作方法的参数直接传入。

按名称绑定

参数绑定方式默认是按照变量名进行绑定，例如，我们给Blog控制器定义了两个操作方法 `read` 和 `archive` 方法，由于 `read` 操作需要指定一个 `id` 参数，`archive` 方法需要指定年份（`year`）和月份（`month`）两个参数，那么我们可以如下定义：

```
namespace app\index\Controller;

class Blog
{
    public function read($id)
    {
        return 'id='.$id;
    }

    public function archive($year='2016',$month='01')
    {
        return 'year='.$year.'&month='.$month;
    }
}
```

注意这里的操作方法并没有具体的业务逻辑，只是简单的示范。

URL的访问地址分别是：

```
http://serverName/index.php/index/blog/read/id/5
http://serverName/index.php/index/blog/archive/year/2016/month/06
```

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

变量名绑定不一定由访问URL决定，路由地址也能起到相同的作用

输出的结果依次是：

```
id=5
year=2016&month=06
```

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

```
http://serverName/index.php/index/blog/archive/month/06/year/2016
```

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/index/blog/read/
```

那么会抛出下面的异常提示：`参数错误:id`

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

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

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

```
id=0
```

始终给操作方法的参数定义默认值是一个避免报错的好办法

按顺序绑定

还可以支持按照URL的参数顺序进行绑定的方式，合理规划URL参数的顺序绑定对简化URL地址可以起到一定的帮助。

还是上面的例子，控制器不变，还是使用：

```
namespace app\index\Controller;

class Blog
{
    public function read($id)
    {
        return 'id='.$id;
    }

    public function archive($year='2016',$month='01')
    {
        return 'year='.$year.'&month='.$month;
    }
}
```

我们在配置文件中添加配置参数如下：

```
// URL参数方式改成顺序解析
'url_param_type'      => 1,
```

接下来，访问下面的URL地址：

```
http://serverName/index.php/index/blog/read/5
http://serverName/index.php/index/blog/archive/2016/06
```

输出的结果依次是：

```
id=5
year=2016&month=06
```

按参数顺序绑定的话，参数的顺序不能随意调整，如果访问：

```
http://serverName/index.php/index/blog/archive/06/2016
```

最后的输出结果则变成：

```
id=5
year=06&month=2016
```

注意

按顺序绑定参数的话，操作方法的参数只能使用URL pathinfo变量，而不能使用get或者post变量。

参数绑定有一个特例，如果你的操作方法中定义有 `Request` 对象作为参数的话，无论参数位置在哪里，都会自动注入，而不需要进行参数绑定。

注入请求对象

控制器的操作方法中如果需要调用请求对象 `Request` 的话，可以在方法中定义 `Request` 类型的参数，并且参数顺序无关，例如：

```
namespace app\index\controller;

use think\Request;

class Index
{
    public function hello(Request $request)
    {
        return 'Hello,' . $request->param('name') . '!!';
    }
}
```

访问URL地址的时候 无需传入 `request` 参数，系统会自动注入当前的 `Request` 对象实例到该参数。

如果继承了系统的 `Controller` 类的话，也可以直接调用 `request` 属性，例如：

```
<?php
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function hello()
    {
        return 'Hello,'.$this->request->param('name');
    }
}
```

HTTP头信息

可以使用Request对象的header方法获取当前请求的HTTP 请求头信息，例如：

```
$info = Request::instance()->header();  
echo $info['accept'];  
echo $info['accept-encoding'];  
echo $info['user-agent'];
```

也可以直接获取某个请求头信息，例如：

```
$agent = Request::instance()->header('user-agent');
```

HTTP请求头信息的名称不区分大小写，并且 `_` 会自动转换为 `-`，所以下面的写法都是等效的：

```
$agent = Request::instance()->header('user-agent');  
$agent = Request::instance()->header('User-Agent');  
$agent = Request::instance()->header('USER_AGENT');
```

方法注入

如果你需要在 `Request` 请求对象中添加自己的方法，可以使用 `Request` 对象的方法注入功能，例如：

```
// 通过hook方法注入动态方法
Request::hook('user', 'getUserInfo');
```

`getUserInfo` 函数定义如下

```
function getUserInfo(Request $request, $userId)
{
    // 根据$userId获取用户信息
    return $info;
}
```

接下来，我们可以直接在控制器中使用：

```
public function index()
{
    $info = Request::instance()->user($userId);
}
```


属性注入

可以动态注入当前 `Request` 对象的属性，方法：

```
// 动态绑定属性
Request::instance()->bind('user', new User);
// 或者使用
Request::instance()->user = new User;
```

获取绑定的属性使用下面的方式：

```
Request::instance()->user;
```

如果控制器注入请求对象的话，也可以直接使用

```
$this->request->user;
```

或者使用助手函数：

```
request()->user;
```

伪静态

URL伪静态通常是为了满足更好的SEO效果，ThinkPHP支持伪静态URL设置，可以通过设置 `url_html_suffix` 参数随意在URL的最后增加你想要的静态后缀，而不会影响当前操作的正常执行。例如，我们设置

```
'url_html_suffix' => 'shtml'
```

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

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

默认情况下，伪静态的设置为 `html`，如果我们设置伪静态后缀为空字符串，

```
'url_html_suffix'=>''
```

则支持所有的静态后缀访问，如果要获取当前的伪静态后缀，可以使用 `Request` 对象的 `ext` 方法。

例如：

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

都可以正常访问。

我们可以在控制器的操作方法中获取当前访问的伪静态后缀，例如：

```
$ext = Request::instance()->ext();
```

如果希望支持多个伪静态后缀，可以直接设置如下：

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

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

如果要关闭伪静态访问，可以设置

```
// 关闭伪静态后缀访问
'url_html_suffix' => false,
```

关闭伪静态访问后，不再支持伪静态方式的URL访问，并且伪静态后缀将会被解析为最后一个参数的值，例如：

```
http://serverName/index/blog/read/id/3.html
```

最终的id参数的值将会变成 `3.html`。

数据库

新版的数据库进行了重构，主要特性包括：

- 类拆分为Connection（连接器）/Query（查询器）/Builder（SQL生成器）
- 新的查询语法
- 闭包查询和闭包事务
- Query对象查询
- 链式操作
- 数据分批处理
- 数据库SQL执行监听

连接数据库

ThinkPHP内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的Db类进行操作，而无需针对不同的数据库写不同的代码和底层实现，Db类会自动调用相应的数据库驱动来处理。采用PDO方式，目前包含了Mysql、SqlServer、PgSQL、Sqlite等数据库的支持。

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。

一、配置文件定义

常用的配置方式是在应用目录或者模块目录下面的 `database.php` 中添加下面的配置参数：

```
return [
    // 数据库类型
    'type'          => 'mysql',
    // 数据库连接DSN配置
    'dsn'           => '',
    // 服务器地址
    'hostname'      => '127.0.0.1',
    // 数据库名
    'database'      => 'thinkphp',
    // 数据库用户名
    'username'      => 'root',
    // 数据库密码
    'password'      => '',
    // 数据库连接端口
    'hostport'      => '',
    // 数据库连接参数
    'params'        => [],
    // 数据库编码默认采用utf8
    'charset'       => 'utf8',
    // 数据库表前缀
    'prefix'        => 'think_',
    // 数据库调试模式
    'debug'         => false,
    // 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)
    'deploy'        => 0,
    // 数据库读写是否分离 主从式有效
    'rw_separate'   => false,
    // 读写分离后 主服务器数量
    'master_num'    => 1,
    // 指定从服务器序号
    'slave_no'      => '',
    // 是否严格检查字段是否存在
    'fields_strict' => true,
];
```

`type` 参数支持命名空间完整定义，不带命名空间定义的话，默认采用 `\think\db\connector` 作为命名空间，如果使用应用自己扩展的数据库驱动，可以配置为：

```
// 数据库类型
'type'          => '\org\db\Mysql',
```

表示数据库的连接器采用 `\org\db\Mysql` 类作为数据库连接驱动，而不是默认的 `\think\db\connector\Mysql`。

每个模块可以设置独立的数据库连接参数，并且相同的配置参数可以无需重复设置，例如我们可以在admin模块的database.php配置文件中定义：

```
return [
    // 服务器地址
    'hostname'    => '192.168.1.100',
    // 数据库名
    'database'    => 'admin',
];
```

表示admin模块的数据库地址改成 `192.168.1.100`，数据库名改成 `admin`，其它的连接参数和应用的 `database.php` 中的配置一样。

连接参数

可以针对不同的连接需要添加数据库的连接参数（具体的连接参数可以参考PHP手册），内置采用的参数包括如下：

```
PDO::ATTR_CASE           => PDO::CASE_NATURAL,
PDO::ATTR_ERRMODE        => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_ORACLE_NULLS   => PDO::NULL_NATURAL,
PDO::ATTR_STRINGIFY_FETCHES => false,
PDO::ATTR_EMULATE_PREPARES => false,
```

在database中设置的params参数中的连接配置将会和内置的设置参数合并，如果需要使用长连接，并且返回数据库的小写列名，可以采用下面的方式定义：

```
'params' => [
    \PDO::ATTR_PERSISTENT => true,
    \PDO::ATTR_CASE       => \PDO::CASE_LOWER,
],
```

你可以在params里面配置任何PDO支持的连接参数。

二、方法配置

我们可以在调用Db类的时候动态定义连接信息，例如：

```
Db::connect([
    // 数据库类型
    'type'        => 'mysql',
    // 数据库连接DSN配置
    'dsn'         => '',
    // 服务器地址
    'hostname'    => '127.0.0.1',
```

```
// 数据库名
'database'    => 'thinkphp',
// 数据库用户名
'username'    => 'root',
// 数据库密码
'password'    => '',
// 数据库连接端口
'hostport'    => '',
// 数据库连接参数
'params'      => [],
// 数据库编码默认采用utf8
'charset'     => 'utf8',
// 数据库表前缀
'prefix'      => 'think_',
]);
```

或者使用字符串方式：

```
Db::connect('mysql://root:1234@127.0.0.1:3306/thinkphp#utf8');
```

字符串连接的定义格式为：

数据库类型://用户名:密码@数据库地址:数据库端口/数据库名#字符集

注意：字符串方式可能无法定义某些参数，例如前缀和连接参数。

如果我们已经在**应用配置文件**（注意这里不是数据库配置文件）中配置了额外的数据库连接信息，例如：

```
//数据库配置1
'db_config1' => [
    // 数据库类型
    'type'      => 'mysql',
    // 服务器地址
    'hostname'  => '127.0.0.1',
    // 数据库名
    'database'  => 'thinkphp',
    // 数据库用户名
    'username'  => 'root',
    // 数据库密码
    'password'  => '',
    // 数据库编码默认采用utf8
    'charset'   => 'utf8',
    // 数据库表前缀
    'prefix'    => 'think_',
],
//数据库配置2
'db_config2' => 'mysql://root:1234@localhost:3306/thinkphp#utf8';
```

我们可以改成

```
Db::connect('db_config1');
Db::connect('db_config2');
```

三、模型类定义

如果在某个模型类里面定义了 `connection` 属性的话，则该模型操作的时候会自动连接给定的数据库连接，而不是配置文件中设置的默认连接信息，通常用于某些数据表位于当前数据库连接之外的其它数据库，例如：

```
//在模型里单独设置数据库连接信息
namespace app\index\model;

use think\Model;

class User extends Model
{
    protected $connection = [
        // 数据库类型
        'type' => 'mysql',
        // 数据库连接DSN配置
        'dsn' => '',
        // 服务器地址
        'hostname' => '127.0.0.1',
        // 数据库名
        'database' => 'thinkphp',
        // 数据库用户名
        'username' => 'root',
        // 数据库密码
        'password' => '',
        // 数据库连接端口
        'hostport' => '',
        // 数据库连接参数
        'params' => [],
        // 数据库编码默认采用utf8
        'charset' => 'utf8',
        // 数据库表前缀
        'prefix' => 'think_',
    ];
}
```

也可以采用DSN字符串方式定义，例如：

```
//在模型里单独设置数据库连接信息
namespace app\index\model;

use think\Model;

class User extends Model
{
    //或者使用字符串定义
    protected $connection = 'mysql://root:1234@127.0.0.1:3306/thinkphp#utf8';
}
```

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。

基本使用

配置了数据库连接信息后，我们就可以直接使用数据库运行原生SQL操作了，支持 `query`（查询操作）和 `execute`（写入操作）方法，并且支持参数绑定。

```
Db::query('select * from think_user where id=?',[8]);  
Db::execute('insert into think_user (id, name) values (?, ?)',[8,'thinkphp']);
```

也支持命名占位符绑定，例如：

```
Db::query('select * from think_user where id=:id',['id'=>8]);  
Db::execute('insert into think_user (id, name) values (:id, :name)',['id'=>8,'name'=>'thinkphp']);
```

可以使用多个数据库连接，使用

```
Db::connect($config)->query('select * from think_user where id=:id',['id'=>8]);
```

`config`是一个单独的数据库配置，支持数组和字符串，也可以是一个数据库连接的配置参数名。

查询构造器

[查询数据](#)

[添加数据](#)

[更新数据](#)

[删除数据](#)

[查询方法](#)

[查询语法](#)

[链式操作](#)

[聚合查询](#)

[时间查询](#)

[高级查询](#)

[视图查询](#)

[子查询](#)

[原生查询](#)

查询数据

基本查询

查询一个数据使用：

```
// table方法必须指定完整的数据表名
Db::table('think_user')->where('id',1)->find();
```

find 方法查询结果不存在，返回 null

查询数据集使用：

```
Db::table('think_user')->where('status',1)->select();
```

select 方法查询结果不存在，返回空数组

如果设置了数据表前缀参数的话，可以使用

```
Db::name('user')->where('id',1)->find();
Db::name('user')->where('status',1)->select();
```

在 `find` 和 `select` 方法之前可以使用所有的链式操作方法。

默认情况下，`find`和`select`方法返回的都是数组。

助手函数

系统提供了一个 `db` 助手函数，可以更方便的查询：

```
db('user')->where('id',1)->find();
db('user')->where('status',1)->select();
```

注意：使用`db`助手函数默认每次都会重新连接数据库，而使用 `Db::name` 或者 `Db::table` 方法的话都是单例的。`db`函数如果需要采用相同的链接，可以传入第三个参数，例如：

```
db('user',[],false)->where('id',1)->find();
db('user',[],false)->where('status',1)->select();
```

上面的方式会使用同一个数据库连接，第二个参数为数据库的连接参数，留空表示采用数据库配置文件的配置。

使用Query对象或闭包查询

或者使用查询对象进行查询，例如：

```
$query = new \think\db\Query();
$query->table('think_user')->where('status',1);
Db::find($query);
Db::select($query);
```

或者直接使用闭包函数查询，例如：

```
Db::select(function($query){
    $query->table('think_user')->where('status',1);
});
```

值和列查询

查询某个字段的值可以用

```
// 返回某个字段的值
Db::table('think_user')->where('id',1)->value('name');
```

value 方法查询结果不存在，返回 false

查询某一列的值可以用

```
// 返回数组
Db::table('think_user')->where('status',1)->column('name');
// 指定索引
Db::table('think_user')->where('status',1)->column('name','id');
```

column 方法查询结果不存在，返回空数组

数据集分批处理

如果你需要处理成千上百条数据库记录，可以考虑使用chunk方法，该方法一次获取结果集的一小块，然后填充每一小块数据到要处理的闭包，该方法在编写处理大量数据库记录的时候非常有用。

比如，我们可以全部用户表数据进行分批处理，每次处理 100 个用户记录：

```
Db::table('think_user')->chunk(100, function($users) {
    foreach ($users as $user) {
        //
    }
});
// 或者交给回调方法myUserIterator处理
Db::table('think_user')->chunk(100, 'myUserIterator');
```

你可以通过从闭包函数中返回false来中止对数据集的处理：

```
Db::table('think_user')->chunk(100, function($users) {  
    // 处理结果集...  
    return false;  
});
```

也支持在chunk方法之前调用其它的查询方法，例如：

```
Db::table('think_user')->where('score', '>', 80)->chunk(100, function($users) {  
    foreach ($users as $user) {  
        //  
    }  
});
```

添加数据

添加一条数据

使用 `Db` 类的 `insert` 方法向数据库提交数据

```
$data = ['foo' => 'bar', 'bar' => 'foo'];  
Db::table('think_user')->insert($data);
```

如果你在 `database.php` 配置文件中配置了数据库前缀(`prefix`), 那么可以直接使用 `Db` 类的 `name` 方法提交数据

```
Db::name('user')->insert($data);
```

`insert` 方法添加数据成功返回添加成功的条数, `insert` 正常情况返回 1

添加数据后如果需要返回新增数据的自增主键, 可以使用 `insertGetId` 方法

```
Db::name('user')->insertGetId($data);
```

`insertGetId` 方法添加数据成功返回添加数据的自增主键

添加多条数据

添加多条数据直接向 `Db` 类的 `insertAll` 方法传入需要添加的数据即可

```
$data = [  
    ['foo' => 'bar', 'bar' => 'foo'],  
    ['foo1' => 'bar1', 'bar1' => 'foo1'],  
    ['foo2' => 'bar2', 'bar2' => 'foo2']  
];  
Db::name('user')->insertAll($data);
```

`insertAll` 方法添加数据成功返回添加成功的条数

助手函数

```
// 添加单条数据  
db('user')->insert($data);  
  
// 添加多条数据  
db('user')->insertAll($list);
```


更新数据

更新数据表中的数据

```
Db::table('think_user')
    ->where('id', 1)
    ->update(['name' => 'thinkphp']);
```

如果数据中包含主键，可以直接使用：

```
Db::table('think_user')
    ->update(['name' => 'thinkphp', 'id'=>1]);
```

update 方法返回影响数据的条数，没修改任何数据返回 0

如果要更新的数据需要使用 SQL 函数或者其它字段，可以使用下面的方式：

```
Db::table('think_user')
    ->where('id', 1)
    ->update([
        'login_time' => ['exp', 'now()'],
        'login_times' => ['exp', 'login_times+1'],
    ]);
```

更新某个字段的值：

```
Db::table('think_user')
    ->where('id', 1)
    ->setField('name', 'thinkphp');
```

setField 方法返回影响数据的条数，没修改任何数据字段返回 0

自增或自减一个字段的值

setInc/setDec 如不加第二个参数，默认值为1

```
// score 字段加 1
Db::table('think_user')
    ->where('id', 1)
    ->setInc('score');
// score 字段加 5
Db::table('think_user')
    ->where('id', 1)
    ->setInc('score', 5);
// score 字段减 1
Db::table('think_user')
    ->where('id', 1)
```

```
->setDec('score');  
// score 字段减 5  
Db::table('think_user')  
    ->where('id', 1)  
    ->setDec('score', 5);
```

setInc/setDec支持延时更新，如果需要延时更新则传入第三个参数

下例中延时10秒，给score字段增加1

```
Db::table('think_user')->where('id', 1)->setInc('score', 1, 10);
```

setInc/setDec 方法返回影响数据的条数

助手函数

```
// 更新数据表中的数据  
db('user')->where('id',1)->update(['name' => 'thinkphp']);  
// 更新某个字段的值  
db('user')->where('id',1)->setField('name', 'thinkphp');  
// 自增 score 字段  
db('user')->where('id', 1)->setInc('score');  
// 自减 score 字段  
db('user')->where('id', 1)->setDec('score');
```

删除数据

删除数据表中的数据

```
// 根据主键删除
Db::table('think_user')->delete(1);
Db::table('think_user')->delete([1,2,3]);

// 条件删除
Db::table('think_user')->where('id',1)->delete();
Db::table('think_user')->where('id','<',10)->delete();
```

delete 方法返回影响数据的条数，没有删除返回 0

助手函数

```
// 根据主键删除
db('user')->delete(1);
// 条件删除
db('user')->where('id',1)->delete();
```

查询方法

条件查询方法

where 方法

可以使用 where 方法进行 AND 条件查询：

```
Db::table('think_user')
->where('name','like','%thinkphp')
->where('status',1)
->find();
```

多字段相同条件的 AND 查询可以简化为如下方式：

```
Db::table('think_user')
->where('name&title','like','%thinkphp')
->find();
```

whereOr 方法

使用 whereOr 方法进行 OR 查询：

```
Db::table('think_user')
->where('name','like','%thinkphp')
->whereOr('title','like','%thinkphp')
->find();
```

多字段相同条件的 OR 查询可以简化为如下方式：

```
Db::table('think_user')
->where('name|title','like','%thinkphp')
->find();
```

getTableInfo 方法

使用getTableInfo可以获取表信息，信息类型 包括 fields,type,bind pk，以数组的形式展示，可以指定某个信息进行获取

```
// 获取`think_user`表所有信息
Db::getTableInfo('think_user');
// 获取`think_user`表所有字段
Db::getTableInfo('think_user', 'fields');
// 获取`think_user`表所有字段的类型
Db::getTableInfo('think_user', 'type');
// 获取`think_user`表的主键
Db::getTableInfo('think_user', 'pk');
```


查询语法

表达式查询

查询表达式支持大部分的SQL查询语法，也是 ThinkPHP 查询语言的精髓，查询表达式的使用格式：

```
where('字段名','表达式','查询条件');
whereOr('字段名','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 |
| [NOT] NULL | 查询字段是否（不）是NULL |
| [NOT] EXISTS | EXISTS查询 |
| EXP | 表达式查询，支持SQL语法 |
| > time | 时间比较 |
| < time | 时间比较 |
| between time | 时间比较 |
| notbetween time | 时间比较 |

表达式查询的用法示例如下：

EQ ：等于（=）

例如：

```
where('id','eq',100);
where('id','=',100);
```

和下面的查询等效

```
where('id',100);
```

表示的查询条件就是 `id = 100`

NEQ：不等于 ()

例如：

```
where('id','neq',100);  
where('id','<>',100);
```

表示的查询条件就是 `id <> 100`

GT：大于 (>)

例如：

```
where('id','gt',100);  
where('id','>',100);
```

表示的查询条件就是 `id > 100`

EGT：大于等于 (>=)

例如：

```
where('id','egt',100);  
where('id','>=',100);
```

表示的查询条件就是 `id >= 100`

LT：小于 (<)

例如：

```
where('id','lt',100);  
where('id','<',100);
```

表示的查询条件就是 `id < 100`

ELT：小于等于 (<=)

例如：

```
where('id','elt',100);  
where('id','<=',100);
```

表示的查询条件就是 `id <= 100`

[NOT] LIKE：同sql的LIKE

例如：

```
where('name','like','thinkphp%');
```

查询条件就变成 `name like 'thinkphp%'`

[NOT] BETWEEN：同sql的[not] between

查询条件支持字符串或者数组，例如：

```
where('id','between','1,8');
```

和下面的等效：

```
where('id','between',[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：

```
where('id','not in','1,5,8');
```

和下面的等效：

```
where('id','not in',[1,5,8]);
```

查询条件就变成 `id NOT IN (1,5, 8)`

[NOT] IN 查询支持使用闭包方式

[NOT] NULL：

查询字段是否（不）是 `Null`，例如：

```
where('name', null);  
where('title','null');  
where('name','not null');
```

如果你需要查询一个字段的值为字符串 `null` 或者 `not null`，应该使用：

```
where('title','=', 'null');  
where('name','=', 'not null');
```


EXP : 表达式

支持更复杂的查询情况 例如：

```
where('id','in','1,3,8');
```

可以改成：

```
where('id','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。

链式操作

数据库提供的链式操作方法，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：

```
Db::table('think_user')
    ->where('status',1)
    ->order('create_time')
    ->limit(10)
    ->select();
```

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

```
Db::table('think_user')
    ->order('create_time')
    ->limit(10)
    ->where('status',1)
    ->select();
```

其实不仅仅是查询方法可以使用连贯操作，包括所有的CURD方法都可以使用，例如：

```
Db::table('think_user')
    ->where('id',1)
    ->field('id,name,email')
    ->find();
Db::table('think_user')
    ->where('status',1)
    ->where('id',1)
    ->delete();
```

链式操作在完成查询后会自动清空链式操作的所有传值。简而言之，链式操作的结果不会带入后面的其它查询。

系统支持的链式操作方法有：

| 连贯操作 | 作用 | 支持的参数类型 |
|------------|---------------|-----------|
| where* | 用于AND查询 | 字符串、数组和对象 |
| whereOr* | 用于OR查询 | 字符串、数组和对象 |
| wheretime* | 用于时间日期的快捷查询 | 数组 字符串 |
| table | 用于定义要操作的数据表名称 | 字符串和数组 |
| alias | 用于给当前数据表定义别名 | 字符串 |

| | | |
|---------------|---------------------|-----------|
| field* | 用于定义要查询的字段（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 |
| view | 用于视图查询 | 字符串、数组 |
| distinct | 用于查询的distinct支持 | 布尔值 |
| lock | 用于数据库的锁机制 | 布尔值 |
| cache | 用于查询缓存 | 支持多个参数 |
| relation | 用于关联查询 | 字符串 |
| with | 用于关联预载入 | 字符串、数组 |
| bind* | 用于数据绑定操作 | 数组或多个参数 |
| comment | 用于SQL注释 | 字符串 |
| force | 用于数据集的强制索引 | 字符串 |
| master | 用于设置主服务器读取数据 | 布尔值 |
| strict | 用于设置是否严格检测字段名是否存在 | 布尔值 |
| sequence | 用于设置Pgsql的自增序列名 | 字符串 |
| failException | 用于设置没有查询到数据是否抛出异常 | 布尔值 |
| partition | 用于设置分表信息 | 数组 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方法的用法是ThinkPHP查询语言的精髓，也是ThinkPHP ORM的重要组成部分和亮点所在，可以完成包括普通查询、表达式查询、快捷查询、区间查询、组合查询在内的查询操作。where方法的参数支持字符串和数组，虽然也可以使用对象但并不建议。

表达式查询

新版的表达式查询采用全新的方式，查询表达式的使用格式：

```
Db::table('think_user')
    ->where('id','>',1)
    ->where('name','thinkphp')
    ->select();
```

更多的表达式查询语法，可以参考[查询语法](#)部分。

数组条件

可以通过数组方式批量设置查询条件。

普通查询

最简单的数组查询方式如下：

```
$map['name'] = 'thinkphp';
$map['status'] = 1;
// 把查询条件传入查询方法
Db::table('think_user')->where($map)->select();

// 助手函数
db('user')->where($map)->select();
```

最后生成的SQL语句是

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

表达式查询

可以在数组条件中使用查询表达式，例如：

```
$map['id'] = ['>',1];
$map['mail'] = ['like','%thinkphp@qq.com%'];
Db::table('think_user')->where($map)->select();
```

字符串条件

使用字符串条件直接查询和操作，例如：

```
Db::table('think_user')->where('type=1 AND status=1')->select();
```

最后生成的SQL语句是

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

使用字符串条件的时候，建议配合预处理机制，确保更加安全，例如：

```
Db::table('think_user')->where("id=:id and username=:name")->bind(['id'=>[1, \PDO::PARAM_INT], 'name'=>'thinkphp'])->select();
```

table

table方法主要用于指定操作的数据表。

用法

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

1. 切换操作的数据表；
2. 对多表进行操作；

例如：

```
Db::table('think_user')->where('status>1')->select();
```

也可以在table方法中指定数据库，例如：

```
Db::table('db_name.think_user')->where('status>1')->select();
```

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

```
Db::table('__USER__')->where('status>1')->select();
```

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

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

如果需要对多表进行操作，可以这样使用：

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

为了尽量避免和mysql的关键字冲突，可以建议使用数组方式定义，例如：

```
Db::field('user.name,role.title')  
->table(['think_user'=>'user','think_role'=>'role'])  
->limit(10)->select();
```

使用数组方式定义的优势是可以避免因为表名和关键字冲突而出错的情况。

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

alias

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

示例：

```
Db::table('think_user')->alias('a')->join('__DEPT__ b ON b.user_id= a.id')->select();
```

最终生成的SQL语句类似于：

```
SELECT * FROM think_user a INNER JOIN think_dept b ON b.user_id= a.id
```


field

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

用于查询

指定字段

在查询操作中field方法是使用最频繁的。

```
Db::table('think_user')->field('id,title,content')->select();
```

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

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

可以给某个字段设置别名，例如：

```
Db::table('think_user')->field('id,nickname as name')->select();
```

执行的SQL语句相当于：

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

使用SQL函数

可以在field方法中直接使用函数，例如：

```
Db::table('think_user')->field('id,SUM(score)')->select();
```

执行的SQL相当于：

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

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

使用数组参数

field方法的参数可以支持数组，例如：

```
Db::table('think_user')->field(['id','title','content'])->select();
```

最终执行的SQL和前面用字符串方式是等效的。

数组方式的定义可以为某些字段定义别名，例如：

```
Db::table('think_user')->field(['id','nickname'=>'name'])->select();
```

执行的SQL相当于：

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

对于一些更复杂的字段要求，数组的优势则更加明显，例如：

```
Db::table('think_user')->field(['id','concat(name, '-',id)'=>'truename','LEFT(title,7)'=>'sub_title'])->select();
```

执行的SQL相当于：

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

获取所有字段

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

```
Db::table('think_user')->select();
Db::table('think_user')->field('*')->select();
```

上面的用法是等效的，都相当于执行SQL：

```
SELECT * FROM table
```

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

```
Db::table('think_user')->field(true)->select();
```

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

字段排除

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

```
Db::table('think_user')->field('content',true)->select();
```

则表示获取除了content之外的所有字段，要排除更多的字段也可以：

```
Db::table('think_user')->field('user_id,content',true)->select();  
//或者用  
Db::table('think_user')->field(['user_id','content'],true)->select();
```

注意的是 字段排除功能不支持跨表和join操作。

用于写入

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

```
Db::table('think_user')->field('title,email,content')->insert($data);
```

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

order

order方法属于模型的连贯操作方法之一，用于对操作的结果排序。

用法如下：

```
Db::table('think_user')->where('status=1')->order('id desc')->limit(5)->select();
```

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

支持对多个字段的排序，例如：

```
Db::table('think_user')->where('status=1')->order('id desc,status')->limit(5)->select();
```

如果没有指定desc或者asc排序规则的话，默认为asc。

如果你的字段和mysql关键字有冲突，那么建议采用数组方式调用，例如：

```
Db::table('think_user')->where('status=1')->order(['order','id'=>'desc'])->limit(5)->select();
```

limit

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

限制结果数量

例如获取满足要求的10个用户，如下调用即可：

```
Db::table('think_user')
->where('status=1')
->field('id,name')
->limit(10)
->select();
```

limit方法也可以用于写操作，例如更新满足要求的3条数据：

```
Db::table('think_user')
->where('score=100')
->limit(3)
->update(['level'=>'A']);
```

分页查询

用于文章分页查询是limit方法比较常用的场合，例如：

```
Db::table('think_article')->limit('10,25')->select();
```

表示查询文章数据，从第10行开始的25条数据（可能还取决于where条件和order排序的影响 这个暂且不提）。

你也可以这样使用，作用是一样的：

```
Db::table('think_article')->limit(10,25)->select();
```

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

page

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

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

```
// 查询第一页数据
Db::table('think_article')->limit('0,10')->select();
// 查询第二页数据
Db::table('think_article')->limit('10,10')->select();
```

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

```
// 查询第一页数据
Db::table('think_article')->page('1,10')->select();
// 查询第二页数据
Db::table('think_article')->page('2,10')->select();
```

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

和limit方法一样，page方法也支持2个参数的写法，例如：

```
Db::table('think_article')->page(1,10)->select();
// 和下面的用法等效
Db::table('think_article')->page('1,10')->select();
```

page方法还可以和limit方法配合使用，例如：

```
Db::table('think_article')->limit(25)->page(3)->select();
```

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

```
Db::table('think_article')->page('3,25')->select();
```

group

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

group方法只有一个参数，并且只能使用字符串。

例如，我们都查询结果按照用户id进行分组统计：

```
Db::table('think_user')
  ->field('user_id,username,max(score)')
  ->group('user_id')
  ->select();
```

生成的SQL语句是：


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

也支持对多个字段进行分组，例如：

```
Db::table('think_user')
  ->field('user_id,test_time,username,max(score)')
  ->group('user_id,test_time')
  ->select();
```

生成的SQL语句是：

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



having

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

having方法只有一个参数，并且只能使用字符串，例如：

```
Db::table('think_user')
  ->field('username,max(score)')
  ->group('user_id')
  ->having('count(test_time)>3')
  ->select();
```

生成的SQL语句是：

```
SELECT username,max(score) FROM think_score GROUP BY user_id HAVING count(test_time)>3
```


join

join通常有下面几种类型，不同类型的join操作会影响返回的数据结果。

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

说明

```
object join ( mixed join [, mixed $condition = null [, string $type = 'INNER']] )
```

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

参数

join

要关联的表。

值为字符串时，表示要关联的表名，如果不是以默认的表前缀或__开头，也不含有括号'()'，会自动补全默认的表前缀；

值为一维数组时，['user'=>'u', 'think_']、['user u', 'think_']都是把think_user做表名，u为别名。这里第二个元素为表前缀，如果不指定会读取默认的表前缀；

值为二维数组时表示是多表关联，关联条件和类型都在子数组中指定，此时不能传第二和第三参数。

condition

关联条件。可以为字符串或数组， 为数组时每一个元素都是一个关联条件。

type

关联类型。可以为:INNER、LEFT、RIGHT、FULL，不区分大小写，默认为INNER。

返回值

模型对象

举例

设默认表前缀为think_

```
Db::table('think_artist')
->alias('a')
->join('work w', 'a.id = w.artist_id')
->join('card c', 'a.card_id = c.id')
```

```
->select();
```

```
Db::table('think_artist')
->alias('a')
->join('__WORK__ w', 'a.id = w.artist_id')
->join('__CARD__ c', 'a.card_id = c.id')
->select();
```

```
$join = [
    ['work w', 'a.id=w.artist_id'],
    ['card c', 'a.card_id=c.id'],
];
Db::table('think_user')->alias('a')->join($join)->select();
```

以上三种写法的效果一样，`__WORK__` 和 `__CARD__` 在最终解析的时候会转换为 `think_work` 和 `think_card`。注意：'_表名_'这种方式中间的表名需要用大写

如果不想使用别名，后面的条件就要使用表全名，可以使用下面这种方式

```
Db::table('think_user')->join('__WORK__', '__ARTIST__.id = __WORK__.artist_id')->select(
);
```

默认采用INNER JOIN 方式，如果需要用其他的JOIN方式，可以改成

```
Db::table('think_user')->alias('a')->join('word w', 'a.id = w.artist_id', 'RIGHT')->select(
);
```

如果需要指定word表前缀为abc_，可以改成

```
Db::table('think_user')->alias('a')->join(['word'=>'w', 'abc_'], 'a.id = w.artist_id', 'RIGHT')->select();
```

表名也可以是一个子查询

```
$subsql = Db::table('think_work')->where(['status'=>1])->field('artist_id,count(id) count')->group('artist_id')->buildSql();
Db::table('think_user')->alias('a')->join($subsql. ' w', 'a.artist_id = w.artist_id')->select();
```

因buildSql返回的语句带有()，所以这里不需要在两端再加上()。

union

UNION操作用于合并两个或多个 SELECT 语句的结果集。

使用示例：

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

数组用法：

```
Db::field('name')
  ->table('think_user_0')
  ->union(['field'=>'name','table'=>'think_user_1'])
  ->union(['field'=>'name','table'=>'think_user_2'])
  ->select();
```

或者

```
Db::field('name')
  ->table('think_user_0')
  ->union(['SELECT name FROM think_user_1','SELECT name FROM think_user_2'])
  ->select();
```

支持UNION ALL 操作，例如：

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

或者

```
Db::field('name')
  ->table('think_user_0')
  ->union(['SELECT name FROM think_user_1','SELECT name FROM think_user_2'],true)
  ->select();
```

每个union方法相当于一个独立的SELECT语句。

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

SELECT 语句中的列的顺序必须相同。

distinct

DISTINCT 方法用于返回唯一不同的值。

例如数据库表中有以下数据

| <input type="checkbox"/> Modify | id | user_login | user_pass | user_name | create_time | status |
|---------------------------------|----|------------|----------------------------------|-----------|-------------|--------|
| <input type="checkbox"/> 编辑 | 1 | chunice | 89f0b495890138511edbca8d446aa63e | chunice | 1463709258 | 1 |
| <input type="checkbox"/> 编辑 | 2 | admin | 89f0b495890138511edbca8d446aa63e | admin | 1463709258 | 1 |
| <input type="checkbox"/> 编辑 | 3 | admin | 89f0b495890138511edbca8d446aa63e | admin | 1463709258 | 1 |

以下代码会返回 `user_login` 字段不同的数据

```
Db::table('think_user')->distinct(true)->field('user_login')->select();
```

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

返回以下数组

```
array(2) {  
    [0] => array(1) {  
        ["user_login"] => string(7) "chunice"  
    }  
    [1] => array(1) {  
        ["user_login"] => string(5) "admin"  
    }  
}
```

distinct方法的参数是一个布尔值。

lock

Lock方法是用于数据库的锁机制，如果在查询或者执行操作的时候使用：

```
lock(true);
```

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

cache

cache方法用于查询缓存操作，也是连贯操作方法之一。

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

下面举例说明，例如，我们对find方法使用cache方法如下：

```
Db::table('think_user')->where('id=5')->cache(true)->find();
```

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

默认情况下，缓存有效期是由默认的缓存配置参数决定的，但 **cache** 方法可以单独指定，例如：

```
Db::table('think_user')->cache(true,60)->find();  
// 或者使用下面的方式 是等效的  
Db::table('think_user')->cache(60)->find();
```

表示对查询结果的缓存有效期60秒。

cache方法可以指定缓存标识：

```
Db::table('think_user')->cache('key',60)->find();
```

指定查询缓存的标识可以使得查询缓存更有效率。

这样，在外部就可以通过 **\think\Cache** 类直接获取查询缓存的数据，例如：

```
$result = Db::table('think_user')->cache('key',60)->find();  
$data = \think\Cache::get('key');
```

cache 方法支持设置缓存标签，例如：

```
Db::table('think_user')->cache('key',60,'tagName')->find();
```

缓存自动更新

这里的缓存自动更新是指一旦数据更新或者删除后会自动清理缓存（下次获取的时候会自动重新缓存）。

当你删除或者更新数据的时候，可以使用cache方法手动更新（清除）缓存，例如：

```
Db::table('think_user')->cache('user_data')->select([1,3,5]);  
Db::table('think_user')->cache('user_data')->update(['id'=>1, 'name'=>'thinkphp']);  
Db::table('think_user')->cache('user_data')->select([1,5]);
```

最后查询的数据不会受第一条查询缓存的影响，确保查询和更新或者删除使用相同的缓存标识才能自动清除缓存。

如果使用 **find** 方法并且使用主键查询的情况，不需要指定缓存标识，会自动清理缓存，例如：

```
Db::table('think_user')->cache(true)->find(1);  
Db::table('think_user')->update(['id'=>1, 'name'=>'thinkphp']);  
Db::table('think_user')->cache(true)->find(1);
```

最后查询的数据会是更新后的数据。

comment

COMMENT方法 用于在生成的SQL语句中添加注释内容，例如：

```
Db::table('think_score')->comment('查询考试前十名分数')
    ->field('username,score')
    ->limit(10)
    ->order('score desc')
    ->select();
```

最终生成的SQL语句是：

```
SELECT username,score FROM think_score ORDER BY score desc LIMIT 10 /* 查询考试前十名分数
*/
```

fetchSql

fetchSql用于直接返回SQL而不是执行查询，适用于任何的CURD操作方法。 例如：

```
$result = Db::table('think_user')->fetchSql(true)->find(1);
```

输出result结果为：`SELECT * FROM think_user where id = 1`

force

force 方法用于数据集的强制索引操作，例如：

```
Db::table('think_user')->force('user')->select();
```

对查询强制使用user索引，user必须是数据表实际创建的索引名称。

using

bind

bind方法用于手动参数绑定，大多数情况，无需进行手动绑定，系统会在查询和写入数据的时候自动使用参数绑定。

bind方法用法如下：

```
// 用于查询
Db::table('think_user')
->where('id', ':id')
->where('name', ':name')
->bind(['id'=>[10,\PDO::PARAM_INT], 'name'=>'thinkphp'])
->select();

// 用于写入
Db::table('think_user')
->bind(['id'=>[10,\PDO::PARAM_INT], 'email'=>'thinkphp@qq.com', 'name'=>'thinkphp'])
->where('id', ':id')
->update(['name'=>':name', 'email'=>':email']);
```

partition

partition 方法用于是数据库水平分表

```
partition($data, $field, $rule);  
// $data 分表字段的数据  
// $field 分表字段的名称  
// $rule 分表规则
```

注意：不要使用任何 SQL 语句中会出现的关键字当表名、字段名，例如 order 等。会导致数据模型拼装 SQL 语句语法错误。

partition 方法用法如下：

```
// 用于写入  
$data = [  
    'user_id'    => 110,  
    'user_name' => 'think'  
];  
  
$rule = [  
    'type' => 'mod', // 分表方式  
    'num'  => 10     // 分表数量  
];  
  
Db::name('log')  
    ->partition(['user_id' => 110], "user_id", $rule)  
    ->insert($data);  
  
// 用于查询  
Db::name('log')  
    ->partition(['user_id' => 110], "user_id", $rule)  
    ->where(['user_id' => 110])  
    ->select();
```

strict

strict 方法用于设置是否严格检查字段名，用法如下：

```
// 关闭字段严格检查
Db::name('user')
  ->strict(false)
  ->insert($data);
```

注意，系统默认值是由数据库配置参数 **fields_strict** 决定，因此修改数据库配置参数可以进行全局的严格检查配置，如下：

```
// 关闭严格检查字段是否存在
'fields_strict' => false,
```

如果开启字段严格检查的话，在更新和写入数据库的时候，一旦存在非数据表字段的值，则会抛出异常。

failException

failException 设置查询数据为空时是否需要抛出异常，如果不传入任何参数，默认为开启，用于 **select** 和 **find** 方法，例如：

```
// 数据不存在的话直接抛出异常
Db:name('blog')->where(['status' => 1])->failException()->select();
// 数据不存在返回空数组 不抛异常
Db:name('blog')->where(['status' => 1])->failException(false)->select();
```

或者可以使用更方便的查空报错

```
// 查询多条
Db:name('blog')->where(['status' => 1])->selectOrFail();

// 查询单条
Db:name('blog')->where(['status' => 1])->findOrFail();
```


sequence

`sequence` 方法用于 `pgsql` 数据库指定自增序列名，其它数据库不必使用，用法为：

```
Db::name('user')->sequence('id')->insert(['name'=>'thinkphp']);
```

聚合查询

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：

获取用户数：

```
Db::table('think_user')->count();
// 助手函数
db('user')->count();
```

或者根据字段统计：

```
Db::table('think_user')->count('id');
// 助手函数
db('user')->count('id');
```

获取用户的最大积分：

```
Db::table('think_user')->max('score');
// 助手函数
db('user')->max('score');
```

获取积分大于0的用户的最小积分：

```
Db::table('think_user')->where('score>0')->min('score');
// 助手函数
db('user')->where('score>0')->min('score');
```

获取用户的平均积分：

```
Db::table('think_user')->avg('score');
// 助手函数
db('user')->avg('score');
```

统计用户的总成绩：

```
Db::table('think_user')->sum('score');  
// 助手函数  
db('user')->sum('score');
```

时间查询

时间比较

使用 where 方法

where 方法支持时间比较，例如：

```
// 大于某个时间
where('create_time', '> time', '2016-1-1');
// 小于某个时间
where('create_time', '<= time', '2016-1-1');
// 时间区间查询
where('create_time', 'between time', ['2015-1-1', '2016-1-1']);
```

第三个参数可以传入任何有效的时间表达式，会自动识别你的时间字段类型，支持的时间类型包括 `timestamps`、`datetime`、`date` 和 `int`。

使用 whereTime 方法

whereTime 方法提供了日期和时间字段的快捷查询，示例如下：

```
// 大于某个时间
db('user')
  ->whereTime('birthday', '>=', '1970-10-1')
  ->select();
// 小于某个时间
db('user')
  ->whereTime('birthday', '<', '2000-10-1')
  ->select();
// 时间区间查询
db('user')
  ->whereTime('birthday', 'between', ['1970-10-1', '2000-10-1'])
  ->select();
// 不在某个时间区间
db('user')
  ->whereTime('birthday', 'not between', ['1970-10-1', '2000-10-1'])
  ->select();
```

时间表达式

还提供了更方便的时间表达式查询，例如：

```
// 获取今天的博客
db('blog')
  ->whereTime('create_time', 'today')
  ->select();
// 获取昨天的博客
db('blog')
  ->whereTime('create_time', 'yesterday')
  ->select();
// 获取本周的博客
db('blog')
  ->whereTime('create_time', 'week')
```

```
->select();
// 获取上周的博客
db('blog')
  ->whereTime('create_time', 'last week')
  ->select();
// 获取本月的博客
db('blog')
  ->whereTime('create_time', 'month')
  ->select();
// 获取上月的博客
db('blog')
  ->whereTime('create_time', 'last month')
  ->select();
// 获取今年的博客
db('blog')
  ->whereTime('create_time', 'year')
  ->select();
// 获取去年的博客
db('blog')
  ->whereTime('create_time', 'last year')
  ->select();
```

如果查询当天、本周、本月和今年的时间，还可以简化为：

```
// 获取今天的博客
db('blog')
  ->whereTime('create_time', 'd')
  ->select();
// 获取本周的博客
db('blog')
  ->whereTime('create_time', 'w')
  ->select();
// 获取本月的博客
db('blog')
  ->whereTime('create_time', 'm')
  ->select();
// 获取今年的博客
db('blog')
  ->whereTime('create_time', 'y')
  ->select();
```

高级查询

快捷查询

快捷查询方式是一种**多字段相同查询条件**的简化写法，可以进一步简化查询条件的写法，在多个字段之间用 | 分割表示OR查询，用 & 分割表示AND查询，可以实现下面的查询，例如：

```
Db::table('think_user')
->where('name|title','like','thinkphp%')
->where('create_time&update_time','>',0)
->find();
```

生成的查询SQL是：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' OR `title` LIKE 'thinkphp%' ) AND ( `create_time` > 0 AND `update_time` > 0 ) LIMIT 1
```

快捷查询支持所有的查询表达式。

区间查询

区间查询是一种**同一字段多个查询条件**的简化写法，例如：

```
Db::table('think_user')
->where('name',['like','thinkphp%'],['like','%thinkphp%'])
->where('id',['>',0],['<=','10'],'or')
->find();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' AND `name` LIKE '%thinkphp%' ) AND ( `id` > 0 OR `id` <= 10 ) LIMIT 1
```

区间查询的查询条件必须使用数组定义方式，支持所有的查询表达式。

下面的查询方式是错误的：

```
Db::table('think_user')
->where('name',['like','thinkphp%'],['like','%thinkphp%'])
->where('id',5,['<=','10'],'or')
->find();
```

批量查询

可以进行多个条件的批量条件查询定义，例如：

```
Db::table('think_user')
    ->where([
        'name' => ['like', 'thinkphp%'],
        'title' => ['like', '%thinkphp'],
        'id' => ['>', 0],
        'status' => 1
    ])
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE `name` LIKE 'thinkphp%' AND `title` LIKE '%thinkphp' AND `id` > 0 AND `status` = '1'
```

EXISTS 查询

可以使用 EXISTS 及 NOT EXISTS 查询：

```
Db::table('think_user')
    ->where('name', 'like', '%thinkphp')
    ->where(function($query){
        $query->where('status', 1);
    }, 'exists')
    ->find();

Db::table('think_user')
    ->where('name', 'like', '%thinkphp')
    ->where(function($query){
        $query->where('status', 1);
    }, 'not exists')
    ->find();
```

包括whereOr方法也支持一样的用法：

```
Db::table('think_user')
    ->where('name', 'like', '%thinkphp')
    ->whereOr(function($query){
        $query->where('status', 1);
    }, 'exists')
    ->find();

Db::table('think_user')
    ->where('name', 'like', '%thinkphp')
    ->whereOr(function($query){
        $query->where('status', 1);
    }, 'not exists')
    ->find();
```

闭包查询

```
Db::table('think_user')->select(function($query){
```

```
$query->where('name','thinkphp')
->whereOr('id','>',10);
});
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE `name` = 'thinkphp' OR `id` > 10
```

使用Query对象查询

也可以事先封装Query对象，并传入select方法，例如：

```
$query = new \think\db\Query;
$query->name('user')
->where('name','like','%think%')
->where('id','>',10)
->limit(10);
Db::select($query);
```

如果使用 Query 对象的话，select 方法之前调用的任何的链式操作都是无效。

混合查询

可以结合前面提到的所有方式进行混合查询，例如：

```
Db::table('think_user')
->where('name',['like','thinkphp%'],['like','%thinkphp'])
->where(function($query){
    $query->where('id',['<',10],['>',100],'or');
})
->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' AND `name` LIKE '%thinkphp' ) AND ( `id` < 10 or `id` > 100 )
```

字符串条件查询

对于一些实在复杂的查询，也可以直接使用原生SQL语句进行查询，例如：

```
Db::table('think_user')
->where('id > 0 AND name LIKE "thinkphp%")
->select();
```

为了安全起见，我们可以对字符串查询条件使用参数绑定，例如：

```
Db::table('think_user')
```



```
->where('id > :id AND name LIKE :name ', ['id'=>0, 'name'=>'thinkphp%'])  
->select();
```

视图查询

视图查询可以实现不依赖数据库视图的多表查询，并不需要数据库支持视图，例如：

```
Db::view('User','id,name')
->view('Profile','truename,phone,email','Profile.user_id=User.id')
->view('Score','score','Score.user_id=Profile.id')
->where('score','>',80)
->select();
```

生成的SQL语句类似于：

```
SELECT User.id,User.name,Profile.truename,Profile.phone,Profile.email,Score.score FROM
think_user User INNER JOIN think_profile Profile ON Profile.user_id=User.id INNER JOIN
think_socre Score ON Score.user_id=Profile.id WHERE Score.score > 80
```

注意，视图查询无需调用 `table` 和 `join` 方法，并且在调用 `where` 和 `order` 方法的时候只需要使用字段名而不需要加表名。

可以使用别名：

```
Db::view('User',['id'=>'uid','name'=>'account'])
->view('Profile','truename,phone,email','Profile.user_id=User.id')
->view('Score','score','Score.user_id=Profile.id')
->where('score','>',80)
->select();
```

生成的SQL语句变成：

```
SELECT User.id AS uid,User.name AS account,Profile.truename,Profile.phone,Profile.email
,Score.score FROM think_user User INNER JOIN think_profile Profile ON Profile.user_id=U
ser.id INNER JOIN think_socre Score ON Score.user_id=Profile.id WHERE Score.score > 80
```

如果你的模型名是数据库的关键字，可以指定数据表的别名，例如：

```
Db::view(['User','member'],['id'=>'uid','name'=>'account'])
->view('Profile','truename,phone,email','Profile.user_id=member.id')
->view('Score','score','Score.user_id=Profile.id')
->where('score','>',80)
->select();
```

生成的SQL语句变成：

```
SELECT member.id AS uid,member.name AS account,Profile.truename,Profile.phone,Profile.e
```

```
mail,Score.score FROM think_user member INNER JOIN think_profile Profile ON Profile.user_id=member.id INNER JOIN think_score Score ON Score.user_id=Profile.id WHERE Score.score > 80
```

子查询

首先构造子查询SQL，可以使用下面三种的方式来构建子查询。

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：

```
$subQuery = Db::table('think_user')
    ->field('id,name')
    ->where('id','>',10)
    ->select(false);
```

生成的subQuery结果为：

```
SELECT `id`,`name` FROM `think_user` WHERE `id` > 10
```

2、使用 fetchSql 方法

fetchSql方法表示不进行查询而只是返回构建的SQL语句，并且不仅仅支持select，而是支持所有的CURD查询。

```
$subQuery = Db::table('think_user')
    ->field('id,name')
    ->where('id','>',10)
    ->fetchSql(true)
    ->select();
```

生成的subQuery结果为：

```
SELECT `id`,`name` FROM `think_user` WHERE `id` > 10
```

3、使用 buildSql 构造子查询

```
$subQuery = Db::table('think_user')
    ->field('id,name')
    ->where('id','>',10)
    ->buildSql();
```

生成的subQuery结果为：

```
( SELECT `id`,`name` FROM `think_user` WHERE `id` > 10 )
```

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。

需要注意的是，使用前两种方法需要自行添加‘括号’。

然后使用子查询构造新的查询：

```
Db::table($subQuery.' a')
    ->where('a.name','like','thinkphp')
    ->order('id','desc')
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM ( SELECT `id`,`name` FROM `think_user` WHERE `id` > 10 ) a WHERE a.name LIKE 'thinkphp' ORDER BY `id` desc
```

4、使用闭包构造子查询

IN 和 EXISTS 之类的查询可以直接使用闭包作为子查询，例如：

```
Db::table('think_user')
    ->where('id','IN',function($query){
        $query->table('think_profile')->where('status',1)->field('id');
    })
    ->select();
```

生成的SQL语句是

```
SELECT * FROM `think_user` WHERE `id` IN ( SELECT `id` FROM `think_profile` WHERE `status` = 1 )
```

```
Db::table('think_user')
    ->where(function($query){
        $query->table('think_profile')->where('status',1);
    }, 'exists')
    ->find();
```

生成的SQL语句为

```
SELECT * FROM `think_user` WHERE EXISTS ( SELECT * FROM `think_profile` WHERE `status` = 1 )
```

原生查询

Db 类支持原生 SQL 查询操作，主要包括下面两个方法：

query 方法

query 方法用于执行 SQL 查询操作，如果数据非法或者查询错误则返回false，否则返回查询结果数据集（同 select 方法）。

使用示例：

```
Db::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。

execute 方法

execute用于更新和写入数据的sql操作，如果数据非法或者查询错误则返回false，否则返回影响的记录数。

使用示例：

```
Db::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。

参数绑定

支持在原生查询的时候使用参数绑定，包括问号占位符或者命名占位符，例如：

```
Db::query("select * from think_user where id=? AND status=?", [8, 1]);  
// 命名绑定  
Db::execute("update think_user set name=:name where status=:status", ['name'=>'thinkphp', 'status'=>1]);
```

事务操作

使用事务处理的话，需要数据库引擎支持事务处理。比如 MySQL 的 MyISAM 不支持事务处理，需要使用 InnoDB 引擎。

使用 transaction 方法操作数据库事务，当发生异常会自动回滚，例如：

自动控制事务处理

```
Db::transaction(function(){
    Db::table('think_user')->find(1);
    Db::table('think_user')->delete(1);
});
```

也可以手动控制事务，例如：

```
// 启动事务
Db::startTrans();
try{
    Db::table('think_user')->find(1);
    Db::table('think_user')->delete(1);
    // 提交事务
    Db::commit();
} catch (\Exception $e) {
    // 回滚事务
    Db::rollback();
}
```

注意在事务操作的时候，确保你的数据库连接是相同的。

监听SQL

如果开启数据库的调试模式的话，你可以对数据库执行的任何SQL操作进行监听，使用如下方法：

```
Db::listen(function($sql, $time, $explain){  
    // 记录SQL  
    echo $sql. ' ['. $time. 's]';  
    // 查看性能分析结果  
    dump($explain);  
});
```

默认如果没有注册任何监听操作的话，这些SQL执行会被根据不同的日志类型记录到日志中。

存储过程

5.0支持存储过程，如果我们定义了一个数据库存储过程 `sp_query`，可以使用下面的方式调用：

```
$result = Db::query('call sp_query(8)');
```

返回的是一个二维数组，也可以使用参数绑定，例如：

```
$result = Db::query('call sp_query(?)',[8]);  
// 或者命名绑定  
$result = Db::query('call sp_query(:id)', ['id'=>8]);
```

数据集

数据库的查询结果也就是数据集，默认的配置下，数据集的类型是一个二维数组，我们可以配置成数据集类，就可以支持对数据集更多的对象化操作，需要使用数据集类功能，可以配置数据库的

`resultset_type` 参数如下：

```
return [
    // 数据库类型
    'type' => 'mysql',
    // 数据库连接DSN配置
    'dsn' => '',
    // 服务器地址
    'hostname' => '127.0.0.1',
    // 数据库名
    'database' => 'thinkphp',
    // 数据库用户名
    'username' => 'root',
    // 数据库密码
    'password' => '',
    // 数据库连接端口
    'hostport' => '',
    // 数据库连接参数
    'params' => [],
    // 数据库编码默认采用utf8
    'charset' => 'utf8',
    // 数据库表前缀
    'prefix' => 'think_',
    // 数据集返回类型
    'resultset_type' => 'collection',
];
```

返回的数据集对象是 `think\Collection`，提供了和数组无差别用法，并且另外封装了一些额外的方法。

可以直接使用数组的方式操作数据集对象，例如：

```
// 获取数据集
$users = Db::name('user')->select();
// 直接操作第一个元素
$item = $users[0];
// 获取数据集记录数
$count = count($users);
// 遍历数据集
foreach($users as $user){
    echo $user['name'];
    echo $user['id'];
}
```

需要注意的是，如果要判断数据集是否为空，不能直接使用 `empty` 判断，而必须使用数据集对象的 `isEmpty` 方法判断，例如：

```
$users = Db::name('user')->select();
```

```
if($users->isEmpty()){
    echo '数据集为空';
}
```

Collection 类包含了下列主要方法：

| 方法 | 描述 |
|-----------|----------------------|
| isEmpty | 是否为空 |
| toArray | 转换为数组 |
| all | 所有数据 |
| merge | 合并其它数据 |
| diff | 比较数组，返回差集 |
| flip | 交换数据中的键和值 |
| intersect | 比较数组，返回交集 |
| keys | 返回数据中的所有键名 |
| pop | 删除数据中的最后一个元素 |
| shift | 删除数据中的第一个元素 |
| unshift | 在数据开头插入一个元素 |
| reduce | 通过使用用户自定义函数，以字符串返回数组 |
| reverse | 数据倒序重排 |
| chunk | 数据分隔为多个数据块 |
| each | 给数据的每个元素执行回调 |
| filter | 用回调函数过滤数据中的元素 |
| column | 返回数据中的指定列 |
| sort | 对数据排序 |
| shuffle | 将数据打乱 |
| slice | 截取数据中的一部分 |

如果只是个别数据的查询需要返回数据集对象，则可以使用

```
Db::name('user')
->fetchClass('\think\Collection')
->select();
```

分布式数据库

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类型。

配置 `database.deploy` 为1 可以采用分布式数据库支持。如果采用分布式数据库，定义数据库配置信息的方式如下：

```
//分布式数据库配置定义
return [
    // 启用分布式数据库
    'deploy'    => 1,
    // 数据库类型
    'type'      => 'mysql',
    // 服务器地址
    'hostname'  => '192.168.1.1,192.168.1.2',
    // 数据库名
    'database'  => 'demo',
    // 数据库用户名
    'username'  => 'root',
    // 数据库密码
    'password'  => '',
    // 数据库连接端口
    'hostport'  => '',
]
```

连接的数据库个数取决于 `hostname` 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：

```
'hostport'=>'3306, 3306'
```

和

```
'hostport'=>'3306'
```

等效。

```
'username'=>'user1',
'password'=>'pwd1',
```

和

```
'username'=>'user1,user1',
'password'=>'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：

```
'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 `master_num` 参数，则可以支持多个主服务器写入。其它的都是从数据库的配置信息，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随机进行在从服务器中选择。

还可以设置 `slave_no` 指定某个服务器进行读操作。

如果从数据库连接错误，会自动切换到主数据库连接。

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

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

模型

新版的模型进行了重构，更加对象化操作，包括关联模型的重构，主要特性包括：

- 完全对象式访问
- 支持静态调用（查询）
- 支持读取器/修改器
- 时间戳字段自动写入
- 对象/数组访问
- JSON序列化
- 模型事件触发
- 命名范围
- 类型自动转换
- 数据验证和自动完成
- 关联查询/操作
- 关联预载入

请不要以3.2版本的模型思维来使用5.0的模型，充分理解和掌握新版模型的用法对于熟练掌握TP5有很大的帮助。

定义

模型定义

定义一个User模型类：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
}
```

默认主键为自动识别，如果需要指定，可以设置属性：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    protected $pk = 'uid';
}
```

模型会自动对应数据表，模型类的命名规则是除去表前缀的数据表名称，采用驼峰法命名，并且首字母大写，例如：

| 模型名 | 约定对应数据表（假设数据库的前缀定义是 think_） |
|----------|-----------------------------|
| User | think_user |
| UserType | think_user_type |

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

设置数据表

如果你想指定数据表甚至数据库连接的话，可以使用：

```
namespace app\index\model;

class User extends \think\Model
{
    // 设置当前模型对应的完整数据表名称
    protected $table = 'think_user';

    // 设置当前模型的数据库连接
    protected $connection = [
        // 数据库类型
        'type' => 'mysql',
    ];
}
```

```

// 服务器地址
'hostname'    => '127.0.0.1',
// 数据库名
'database'    => 'thinkphp',
// 数据库用户名
'username'    => 'root',
// 数据库密码
'password'    => '',
// 数据库编码默认采用utf8
'charset'     => 'utf8',
// 数据库表前缀
'prefix'      => 'think_',
// 数据库调试模式
'debug'       => false,
];
}

```

5.0不支持单独设置当前模型的数据表前缀。

设置字段信息

默认情况下，系统会自动获取模型对应数据表的字段信息，如果你希望减少数据库的开销，也可以手动设置数据表的字段信息包括类型，例如：

```

namespace app\index\model;

class User extends \think\Model
{
    // 设置当前模型对应的完整数据表名称
    protected $table = 'think_user';
    // 设置数据表主键
    protected $pk    = 'id';
    // 设置当前数据表的字段信息
    protected $field = [
        'id' => 'int',
        'name', 'email',
    ];
}

```

如果字段类型为字符串类型可以不设置类型。

模型调用

模型类可以使用静态调用或者实例化调用两种方式，例如：

```

// 静态调用
$user = User::get(1);
$user->name = 'thinkphp';
$user->save();

// 实例化模型
$user = new User;
$user->name = 'thinkphp';
$user->save();

```



```
// 使用 Loader 类实例化（单例）  
$user = Loader::model('User');  
  
// 或者使用助手函数`model`  
$user = model('User');  
$user->name= 'thinkphp';  
$user->save();
```

实例化模型类主要用于调用模型的自定义方法，更多用法参考后面的章节内容。

模型初始化

模型初始化

模型同样支持初始化，与控制器的初始化不同的是，模型的初始化是重写 `Model` 的 `initialize`，具体如下

```
namespace app\index\model;

use think\Model;

class Index extends Model
{
    //自定义初始化
    protected function initialize()
    {
        //需要调用`Model`的`initialize`方法
        parent::initialize();
        //TODO:自定义的初始化
    }
}
```

同样也可以使用静态 `init` 方法，需要注意的是 `init` 只在第一次实例化的时候执行，并且方法内需要注意静态调用的规范，具体如下：

```
namespace app\index\model;

use think\Model;

class Index extends Model
{
    //自定义初始化
    protected static function init()
    {
        //TODO:自定义的初始化
    }
}
```

新增

新增数据有多种方式。

添加一条数据

第一种是实例化模型对象后赋值并保存：

```
$user = new User;
$user->name = 'thinkphp';
$user->email = 'thinkphp@qq.com';
$user->save();
```

也可以使用 `data` 方法批量赋值：

```
$user = new User;
$user->data([
    'name' => 'thinkphp',
    'email' => 'thinkphp@qq.com'
]);
$user->save();
```

或者直接在实例化的时候传入数据

```
$user = new User([
    'name' => 'thinkphp',
    'email' => 'thinkphp@qq.com'
]);
$user->save();
```

如果需要过滤非数据表字段的数据，可以使用：

```
$user = new User($_POST);
// 过滤post数组中的非数据表字段数据
$user->allowField(true)->save();
```

如果你通过外部提交赋值给模型，并且希望指定某些字段写入，可以使用：

```
$user = new User($_POST);
// post数组中只有name和email字段会写入
$user->allowField(['name', 'email'])->save();
```

`save`方法新增数据返回的是写入的记录数。

获取自增ID

如果要获取新增数据的自增ID，可以使用下面的方式：

```
$user          = new User;
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
// 获取自增ID
echo $user->id;
```

添加多条数据

支持批量新增，可以使用：

```
$user = new User;
$list = [
    ['name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['name'=>'onethink', 'email'=>'onethink@qq.com']
];
$user->saveAll($list);
```

saveAll方法新增数据返回的是包含新增模型（带自增ID）的数据集（数组）。

saveAll 方法新增数据默认会自动识别数据是需要新增还是更新操作，当数据中存在主键的时候会认为是更新操作，如果你需要带主键数据批量新增，可以使用下面的方式：

```
$user = new User;
$list = [
    ['id'=>1, 'name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['id'=>2, 'name'=>'onethink', 'email'=>'onethink@qq.com'],
];
$user->saveAll($list, false);
```

如果你自己通过遍历批量新增数据，可以参考下面的方法：

```
$user = new User;
$list = [
    ['name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['name'=>'onethink', 'email'=>'onethink@qq.com']
];
foreach($list as $data){
    $user->data($data, true)->isUpdate(false)->save();
}
```

静态方法

还可以直接静态调用 create 方法创建并写入：

```
$user = User::create([
    'name' => 'thinkphp',
    'email' => 'thinkphp@qq.com'
```

```
]);  
echo $user->name;  
echo $user->email;  
echo $user->id; // 获取自增ID
```

和save方法不同的是，create方法返回的是当前模型的对象实例。

助手函数

系统提供了model助手函数用于快速实例化模型，并且使用单例实现，例如：

```
// 使用model助手函数实例化User模型  
$user = model('User');  
// 模型对象赋值  
$user->data([  
    'name' => 'thinkphp',  
    'email' => 'thinkphp@qq.com'  
]);  
$user->save();
```

或者进行批量新增：

```
$user = model('User');  
// 批量新增  
$list = [  
    ['name'=>'thinkphp','email'=>'thinkphp@qq.com'],  
    ['name'=>'onethink','email'=>'onethink@qq.com']  
];  
$user->saveAll($list);
```

更新

查找并更新

在取出数据后，更改字段内容后更新数据。

```
$user = User::get(1);
$user->name      = 'thinkphp';
$user->email      = 'thinkphp@qq.com';
$user->save();
```

直接更新数据

也可以直接带更新条件来更新数据

```
$user = new User;
// save方法第二个参数为更新条件
$user->save([
    'name' => 'thinkphp',
    'email' => 'thinkphp@qq.com'
], ['id' => 1]);
```

上面两种方式更新数据，如果需要过滤非数据表字段的数据，可以使用：

```
$user = new User();
// 过滤post数组中的非数据表字段数据
$user->allowField(true)->save($_POST, ['id' => 1]);
```

如果你通过外部提交赋值给模型，并且希望指定某些字段写入，可以使用：

```
$user = new User();
// post数组中只有name和email字段会写入
$user->allowField(['name', 'email'])->save($_POST, ['id' => 1]);
```

批量更新数据

可以使用saveAll方法批量更新数据，例如：

```
$user = new User;
$list = [
    ['id'=>1, 'name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['id'=>2, 'name'=>'onethink', 'email'=>'onethink@qq.com']
];
$user->saveAll($list);
```

批量更新仅能根据主键值进行更新，其它情况请使用 `foreach` 遍历更新。

如果你自己通过遍历批量更新数据，可以参考下面的方法：

```
$user = new User;
$list = [
    ['id'=>1, 'name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['id'=>2, 'name'=>'onethink', 'email'=>'onethink@qq.com']
];
foreach($list as $data){
    $user->data($data,true)->isUpdate(true)->save();
}
```

通过数据库类更新数据

必要的时候，你也可以使用数据库对象来直接更新数据，但这样就无法使用模型的事件功能。

```
$user = new User;
$user->where('id', 1)
    ->update(['name' => 'thinkphp']);
```

或者使用：

```
$user = new User;
$user->update(['id' => 1, 'name' => 'thinkphp']);
```

如果传入 `update` 的数据包含主键的话，可以无需使用 `where` 方法。

静态方法

模型支持静态方法直接更新数据，例如：

```
User::where('id', 1)
    ->update(['name' => 'thinkphp']);
```

或者使用：

```
User::update(['id' => 1, 'name' => 'thinkphp']);
```

闭包更新

可以通过闭包函数使用更复杂的更新条件，例如：

```
$user = new User;
$user->save(['name' => 'thinkphp'],function($query){
    // 更新status值为1 并且id大于10的数据
    $query->where('status', 1)->where('id', '>', 10);
});
```

自动识别

我们已经看到，模型的新增和更新方法都是 `save` 方法，系统有一套默认的规则来识别当前的数据需要更新还是新增。

- 实例化模型后调用save方法表示新增；
- 查询数据后调用save方法表示更新；
- 调用模型的save方法后表示更新；

如果你的数据操作比较复杂，可以显式的指定当前调用 `save` 方法是新增操作还是更新操作。

显式更新数据：

```
// 实例化模型
$user = new User;
// 显式指定更新数据操作
$user->isUpdate(true)
    ->save(['id' => 1, 'name' => 'thinkphp']);
```

显示新增数据：

```
$user = User::get(1);
$user->name = 'thinkphp';
// 显式指定当前操作为新增操作
$user->isUpdate(false)->save();
```

如果你调用save方法进行多次数据写入的时候，需要注意，第二次save方法的时候必须使用isUpdate(false)，否则会视为更新数据。

删除

删除当前模型

删除模型数据，可以在实例化后调用 `delete` 方法。

```
$user = User::get(1);  
$user->delete();
```

根据主键删除

或者直接调用静态方法

```
User::destroy(1);  
// 支持批量删除多个数据  
User::destroy('1,2,3');  
// 或者  
User::destroy([1,2,3]);
```

条件删除

使用数组进行条件删除，例如：

```
// 删除状态为0的数据  
User::destroy(['status' => 0]);  
});
```

还支持使用闭包删除，例如：

```
User::destroy(function($query){  
    $query->where('id', '>', 10);  
});
```

或者通过数据库类的查询条件删除

```
User::where('id', '>', 10)->delete();
```

查询

获取单个数据

获取单个数据的方法包括：

```
取出主键为1的数据
$user = User::get(1);
echo $user->name;

// 使用数组查询
$user = User::get(['name' => 'thinkphp']);

// 使用闭包查询
$user = User::get(function($query){
    $query->where('name', 'thinkphp');
});
echo $user->name;
```

或者在实例化模型后调用查询方法

```
$user = new User();
// 查询单个数据
$user->where('name', 'thinkphp')
->find();
```

`get` 或者 `find` 方法返回的是当前模型的对象实例，可以使用模型的方法。

获取多个数据

取出多个数据：

```
// 根据主键获取多个数据
$list = User::all('1,2,3');
// 或者使用数组
$list = User::all([1,2,3]);
foreach($list as $key=>$user){
    echo $user->name;
}
// 使用数组查询
$list = User::all(['status'=>1]);
// 使用闭包查询
$list = User::all(function($query){
    $query->where('status', 1)->limit(3)->order('id', 'asc');
});
foreach($list as $key=>$user){
    echo $user->name;
}
```

数组方式和闭包方式的数据查询的区别在于，数组方式只能定义查询条件，闭包方式可以支持更多的连贯

操作，包括排序、数量限制等。

或者在实例化模型后调用查询方法

```
$user = new User();
// 查询数据集
$user->where('name', 'thinkphp')
    ->limit(10)
    ->order('id', 'desc')
    ->select();
```

模型的 `all` 方法或者 `select` 方法返回的是一个包含模型对象的二维数组或者数据集对象。

获取某个字段或者某个列的值

```
// 获取某个用户的积分
User::where('id',10)->value('score');
// 获取某个列的所有值
User::where('status',1)->column('name');
// 以id为索引
User::where('status',1)->column('name','id');
```

注意 `value` 和 `column` 方法返回的不再是一个模型对象实例，而是单纯的值或者某个列的数组。

动态查询

支持动态查询方法，例如：

```
// 根据name字段查询用户
$user = User::getByName('thinkphp');

// 根据email字段查询用户
$user = User::getByEmail('thinkphp@qq.com');
```

通过Query类查询

或者使用数据库的查询方法进行更复杂的查询：

```
User::where('id','>',10)->select();
User::where('name','thinkphp')->find();
```

可以在模型中直接使用所有数据库的链式操作方法。

返回的查询结果是当前模型对应的对象或者包含模型对象的数据集。

数据分批处理

模型也可以支持对返回的数据分批处理，这在处理大量数据的时候非常有用，例如：

```
User::chunk(100,function($users){
    foreach($users as $user){
        // 处理user模型对象
    }
});
```

聚合

| 在模型中也可以调用数据库的聚合方法进行查询，例如： | 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） | |

静态调用：

```
User::count();
User::where('status','>',0)->count();
User::where('status',1)->avg('score');
User::max('score');
```

动态调用：

```
$user = new User;
$user->count();
$user->where('status','>',0)->count();
$user->where('status',1)->avg('score');
$user->max('score');
```

获取器

获取器的作用是在获取数据的字段值后自动进行处理，例如，我们需要对状态值进行转换，可以使用：

```
class User extends Model
{
    public function getStatusAttr($value)
    {
        $status = [-1=>'删除',0=>'禁用',1=>'正常',2=>'待审核'];
        return $status[$value];
    }
}
```

数据表的字段会自动转换为驼峰法，一般 `status` 字段的值采用数值类型，我们可以通过获取器定义，自动转换为字符串描述。

```
$user = User::get(1);
echo $user->status; // 例如输出“正常”
```

获取器还可以定义数据表中不存在的字段，例如：

```
class User extends Model
{
    public function getStatusTextAttr($value,$data)
    {
        $status = [-1=>'删除',0=>'禁用',1=>'正常',2=>'待审核'];
        return $status[$data['status']];
    }
}
```

我们就可以直接使用`status_text`字段的值了，例如：

```
$user = User::get(1);
echo $user->status_text; // 例如输出“正常”
```

如果你定义了获取器的情况下，希望获取数据表中的原始数据，可以使用：

```
$user = User::get(1);
// 通过获取器获取字段
echo $user->status;
// 获取原始字段数据
echo $user->getData('status');
```

修改器

修改器的作用是可以在数据赋值的时候自动进行转换处理，例如：

```
class User extends Model
{
    public function setNameAttr($value)
    {
        return strtolower($value);
    }
}
```

如下代码实际保存到数据库中的时候会转为小写

```
$user = new User();
$user->name = 'THINKPHP';
$user->save();
echo $user->name; // thinkphp
```

也可以进行序列化字段的组装：

```
class User extends Model
{
    public function setNameAttr($value,$data)
    {
        return serialize($data);
    }
}
```

除了赋值的方式可以触发修改器外，还可以用下面的方法触发修改器：

```
$user = new User();
$data['name'] = 'THINKPHP';
$user->data($data, true);
$user->save();
echo $user->name; // thinkphp
```

时间戳

系统支持自动写入创建和更新的时间戳字段，有两种方式配置支持。

第一种方式，是在数据库配置文件中添加全局设置：

```
// 开启自动写入时间戳字段
'auto_timestamp' => true,
```

第二种是直接在单独的模型类里面设置：

```
protected $autoWriteTimestamp = true;
```

如果这两个地方设置为true，默认识别为整型 `int` 类型，如果你的时间字段不是 `int` 类型的话，例如使用 `datetime` 类型的话，可以这样设置：

```
// 开启自动写入时间戳字段
'auto_timestamp' => 'datetime',
```

或者

```
protected $autoWriteTimestamp = 'datetime';
```

字段名默认创建时间字段为 `create_time`，更新时间字段为 `update_time`，支持的字段类型包括 `timestamp/datetime/int`。

写入数据的时候，系统会自动写入 `create_time` 和 `update_time` 字段，而不需要定义修改器，例如：

```
$user = new User();
$user->name = 'THINKPHP';
$user->save();
echo $user->create_time; // 输出类似 1462545582
echo $user->update_time; // 输出类似 1462545582
```

时间戳字段输出的时候一样支持获取器。

如果你的数据表字段不是默认值的话，可以按照下面的方式定义：

```
class User extends Model
{
    // 定义时间戳字段名
    protected $createTime = 'create_at';
```



```
protected $updateTime = 'update_at';
}
```

下面是修改字段后的输出代码：

```
$user = new User();
$user->name = 'THINKPHP';
$user->save();
echo $user->create_at; // 输出类似 1462545612
echo $user->update_at; // 输出类似 1462545612
```

如果你只需要使用 `create_time` 字段而不需要自动写入 `update_time`，则可以单独设置关闭某个字段，例如：

```
class User extends Model
{
    // 关闭自动写入update_time字段
    protected $updateTime = false;
}
```

如果不需要任何自动写入的时间戳字段的话，可以关闭时间戳自动写入功能，设置如下：

```
class User extends Model
{
    // 关闭自动写入时间戳
    protected $autoWriteTimestamp = false;
}
```

如果是关闭全局的自动时间写入，则可以使用：

```
// 关闭全局自动写入时间字段
'auto_timestamp' => false,
```

软删除

在实际项目中，对数据频繁使用删除操作会导致性能问题，软删除的作用就是把数据加上删除标记，而不是真正的删除，同时也便于需要的时候进行数据的恢复。

要使用软删除功能，需要引入 `SoftDelete` trait，例如 `User` 模型按照下面的定义就可以使用软删除功能：

```
namespace app\index\model;

use think\Model;
use traits\model\SoftDelete;

class User extends Model
{
    use SoftDelete;
    protected static $deleteTime = 'delete_time';
}
```

`deleteTime` 属性用于定义你的软删除标记字段，`ThinkPHP5` 的软删除功能使用时间戳类型（数据表默认为 `Null`），用于记录数据的删除时间。

可以用类型转换指定软删除字段的类型。

定义好模型后，我们就可以使用：

```
// 软删除
User::destroy(1);
// 真实删除
User::destroy(1,true);
$user = User::get(1);
// 软删除
$user->delete();
// 真实删除
$user->delete(true);
```

默认情况下查询的数据不包含软删除数据，如果需要包含软删除的数据，可以使用下面的方式查询：

```
User::withTrashed()->find();
User::withTrashed()->select();
```

如果仅仅需要查询软删除的数据，可以使用：

```
User::onlyTrashed()->find();
User::onlyTrashed()->select();
```

软删除

类型转换

支持给字段设置类型自动转换，会在写入和读取的时候自动进行类型转换处理，例如：

```
class User extends Model
{
    protected $type = [
        'status'    => 'integer',
        'score'     => 'float',
        'birthday'  => 'datetime',
        'info'      => 'array',
    ];
}
```

下面是一个类型自动转换的示例：

```
$user = new User;
$user->status = '1';
$user->score = '90.50';
$user->birthday = '2015/5/1';
$user->info = ['a'=>1, 'b'=>2];
$user->save();
var_dump($user->status); // int 1
var_dump($user->score); // float 90.5;
var_dump($user->birthday); // string '2015-05-01 00:00:00'
var_dump($user->info); // array (size=2) 'a' => int 1  'b' => int 2
```

数据库查询默认取出来的数据都是字符串类型，如果需要转换为其他的类型，需要设置，支持的类型包括如下类型：

integer

设置为integer（整型）后，该字段写入和输出的时候都会自动转换为整型。

float

该字段的值写入和输出的时候自动转换为浮点型。

boolean

该字段的值写入和输出的时候自动转换为布尔型。

array

如果设置为强制转换为 **array** 类型，系统会自动把数组编码为json格式字符串写入数据库，取出来的时候会自动解码。

object

该字段的值在写入的时候会自动编码为json字符串，输出的时候会自动转换为 **stdClass** 对象。

serialize

指定为序列化类型的话，数据会自动序列化写入，并且在读取的时候自动反序列化。

json

指定为 `json` 类型的话，数据会自动 `json_encode` 写入，并且在读取的时候自动 `json_decode` 处理。

timestamp

指定为时间戳字段类型的话，该字段的值在写入时候会自动使用 `strtotime` 生成对应的时间戳，输出的时候会自动转换为 `dateFormat` 属性定义的时间字符串格式，默认的格式为 `Y-m-d H:i:s`，如果希望改变其他格式，可以定义如下：

```
class User extends Model
{
    protected $dateFormat = 'Y/m/d';
    protected $type = [
        'status'    => 'integer',
        'score'     => 'float',
        'birthday'  => 'timestamp',
    ];
}
```

或者在类型转换定义的时候使用：

```
class User extends Model
{
    protected $type = [
        'status'    => 'integer',
        'score'     => 'float',
        'birthday'  => 'timestamp:Y/m/d',
    ];
}
```

然后就可以

```
$user = User::find(1);
echo $user->birthday; // 2015/5/1
```

datetime

和 `timestamp` 类似，区别在于写入和读取数据的时候都会自动处理成时间字符串 `Y-m-d H:i:s` 的格式。

数据完成

数据自动完成指在不需要手动赋值的情况下对字段的值进行处理后写入数据库。

系统支持 `auto`、`insert` 和 `update` 三个属性，可以分别在写入、新增和更新的时候进行字段的自动完成机制，`auto`属性自动完成包含新增和更新操作，例如我们定义 `User` 模型类如下：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    protected $auto = ['name', 'ip'];
    protected $insert = ['status'=>1];
    protected $update = [];

    protected function setNameAttr($value)
    {
        return strtolower($value);
    }

    protected function setIpAttr()
    {
        return request()->ip();
    }
}
```

在新增数据的时候，会对 `name`、`ip` 和 `status` 字段自动完成或者处理。

```
$user = new User;
$user->name = 'ThinkPHP';
$user->save();
echo $user->name; // thinkphp
echo $user->status; // 1
```

在保存操作的时候，会自动处理 `name` 字段的值及完成 `ip` 字段的赋值。

```
$user = User::find(1);
$user->name = 'THINKPHP';
$user->save();
echo $user->name; // thinkphp
echo $user->ip; // 127.0.0.1
```

开发者需要理清“修改器”与“自动完成”的关系。

查询范围

可以对模型的查询和写入操作进行封装，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    protected function scopeThinkphp($query)
    {
        $query->where('name', 'thinkphp')->field('id,name');
    }

    protected function scopeAge($query)
    {
        $query->where('age', '>', 20)->limit(10);
    }
}
```

就可以进行下面的条件查询：

```
// 查找name为thinkphp的用户
User::scope('thinkphp')->get();
// 查找年龄大于20的10个用户
User::scope('age')->all();
// 查找name为thinkphp的用户并且年龄大于20的10个用户
User::scope('thinkphp,age')->all();
```

可以直接使用闭包函数进行查询，例如：

```
User::scope(function($query){
    $query->where('age', '>', 20)->limit(10);
})->all();
```

支持动态调用的方式，例如：

```
$user = new User;
// 查找name为thinkphp的用户
$user->thinkphp()->get();
// 查找年龄大于20的10个用户
$user->age()->all();
// 查找name为thinkphp的用户并且年龄大于20的10个用户
$user->thinkphp()->age()->all();
```

命名范围方法之前不能调用查询的连贯操作方法，必须首先被调用。

全局查询范围

如果你的所有查询都需要一个基础的查询范围，那么可以在模型类里面定义一个静态的 `base` 方法，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    // 定义全局的查询范围
    protected static function base($query)
    {
        $query->where('status',1);
    }
}
```

全局查询范围方法必须定义为 `static` 静态方法。

然后，执行下面的代码：

```
$user = User::get(1);
```

最终的查询条件会是

```
status = 1 AND id = 1
```

如果需要动态关闭/开启全局查询访问，可以使用：

```
// 关闭全局查询范围
User::useGlobalScope(false)->get(1);
// 开启全局查询范围
User::useGlobalScope(true)->get(2);
```


模型分层

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

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

例如在 `index` 模块的设计中需要区分数据层、逻辑层、服务层等不同的模型层，我们可以在模块目录下面创建 `model`、`logic` 和 `service` 目录，把对用户表的所有模型操作分成三层：

- 数据层：`app\index\model\User` 用于定义数据相关的自动验证和自动完成和数据存取接口
- 逻辑层：`app\index\logic\User` 用于定义用户相关的业务逻辑
- 服务层：`app\index\service\User` 用于定义用户相关的服务接口等

三个模型层的定义如下：

`app\index\model\User.php`

```
namespace app\index\model;

use think\Model;

class User extends Model
{
}
```

实例化方法：`\think\Loader::model('User')`

Logic类：`app\index\logic\User.php`

```
namespace app\index\logic;

use think\Model;

class User extends Model
{
}
```

实例化方法：`\think\Loader::model('User', 'logic');`

Service类：`app\index\service\User.php`

```
namespace app\index\service;

use think\Model;

class User extends Model
{
}
```

```
}
```

实例化方法：`\think\Loader::model('User','service');`

数组访问和转换

数组访问

模型对象支持数组方式访问，例如：

```
$user = User::find(1);  
echo $user->name ; // 有效  
echo $user['name'] // 同样有效  
$user->name = 'thinkphp'; // 有效  
$user['name'] = 'thinkphp'; // 同样有效  
$user->save();
```

转换为数组

可以使用 `toArray` 方法将当前的模型实例输出为数组，例如：

```
$user = User::find(1);  
dump($user->toArray());
```

支持设置不输出的字段属性：

```
$user = User::find(1);  
dump($user->hidden(['create_time', 'update_time'])->toArray());
```

数组输出的字段值会经过获取器的处理。d

也可以支持追加其它获取器定义的字段，例如：

```
$user = User::find(1);  
dump($user->append(['status_text'])->toArray());
```

支持设置允许输出的属性，例如：

```
$user = User::find(1);  
dump($user->visible(['id', 'name', 'email'])->toArray());
```

JSON序列化

可以调用模型的 `toJson` 方法进行 **JSON** 序列化

```
$user = User::get(1);  
echo $user->toJson();
```

可以设置无需输出的字段，例如：

```
$user = User::get(1);  
echo $user->hidden(['create_time', 'update_time'])->toJson();
```

或者追加其它的字段：

```
$user = User::get(1);  
echo $user->append(['status_text'])->toJson();
```

设置允许输出的属性：

```
$user = User::get(1);  
echo $user->visible(['id', 'name', 'email'])->toJson();
```

模型对象可以直接被JSON序列化，例如：

```
echo json_encode(User::get(1));
```

输出结果类似于：

```
{"id":"1","name":"","title":"","status":"1","update_time":"1430409600","score":"90.5"}
```

或者也可以直接 `echo` 一个模型对象，例如：

```
echo User::get(1);
```

输出的结果和上面是一样的。

事件

模型类支持 `before_delete`、`after_delete`、`before_write`、`after_write`、`before_update`、`after_update`、`before_insert`、`after_insert` 事件行为，使用方法如下：

```
User::event('before_insert',function($user){
    if($user->status != 1){
        return false;
    }
});
```

注册的回调方法支持传入一个参数（当前的模型对象实例），并且 `before_write`、`before_insert`、`before_update`、`before_delete` 事件方法如果返回false，则不会继续执行。

支持给一个位置注册多个回调方法，例如：

```
User::event('before_insert',function($user){
    if($user->status != 1){
        return false;
    }
});
// 注册回调到beforeInsert函数
User::event('before_insert','beforeInsert');
```

关联

一对一关联

一对一关联定义

定义一对一关联，例如，一个用户都有一个个人资料，我们定义 `User` 模型如下：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile');
    }
}
```

`hasOne` 方法的参数包括：

`hasOne('关联模型名','外键名','主键名',['模型别名定义'],'join类型');`

默认的 `join` 类型为 `INNER`。

关联查找

定义好关联之后，就可以使用下面的方法获取关联数据：

```
$user = User::find(1);
// 输出Profile关联模型的email属性
echo $user->profile->email;
```

默认情况下，我们使用的是 `user_id` 作为外键关联，如果不是的话则需要在关联定义的时候指定，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile','uid');
    }
}
```

设置别名

如果你的模型名和数据库关键字冲突的话，可以设置别名，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile','uid')->setAlias(['user'=>'member']);
    }
}
```

根据关联查询条件查询

可以根据关联条件来查询当前模型对象数据，例如：

```
// 查询有档案记录的用户
$user = User::has('profile')->find();
// 查询年龄是20岁的用户
$user = User::hasWhere('profile',['age'=>'20'])->select();
```

关联新增

```
$user = User::find(1);
// 如果还没有关联数据 则进行新增
$user->profile()->save(['email' => 'thinkphp']);
```

系统会自动把当前模型的主键传入profile模型。

关联更新

和新增一样使用 `save` 方法进行更新关联数据。

```
$user = User::find(1);
$user->profile->email = 'thinkphp';
$user->profile->save();
// 或者
$user->profile->save(['email' => 'thinkphp']);
```

定义相对的关联

我们可以在 `Profile` 模型中定义一个相对的关联关系，例如：

```
namespace app\index\model;

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo('User');
    }
}
```


`belongsTo` 的参数包括：

`belongsTo('关联模型名','外键名','关联表主键名',['模型别名定义'],'join类型');`

默认的关联外键是 `user_id`，如果不是，需要在第二个参数定义

```
namespace app\index\model;

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo('User', 'uid');
    }
}
```

我们就可以根据档案资料来获取用户模型的信息

```
$profile = Profile::find(1);
// 输出User关联模型的属性
echo $profile->user->account;
```

一对多关联

一对多关联

关联定义

一对多关联的情况也比较常见，使用 `hasMany` 方法定义，参数包括：

```
hasMany('关联模型名','外键名','主键名',['模型别名定义']);
```

例如一篇文章可以有多个评论

```
namespace app\index\model;

use think\Model;

class Article extends Model
{
    public function comments()
    {
        return $this->hasMany('Comment');
    }
}
```

同样，也可以定义外键的名称

```
namespace app\index\model;

use think\Model;

class Article extends Model
{
    public function comments()
    {
        return $this->hasMany('Comment','art_id');
    }
}
```

关联查询

我们可以通过下面的方式获取关联数据

```
$article = Article::get(1);
// 获取文章的所有评论
dump($article->comments);
// 也可以进行条件搜索
dump($article->comments()->where('status',1)->select());
```

根据关联条件查询

可以根据关联条件来查询当前模型对象数据，例如：

```
// 查询评论超过3个的文章
$list = Article::has('comments', '>', 3)->select();
// 查询评论状态正常的文章
$list = Article::hasWhere('comments', ['status'=>1])->select();
```

关联新增

```
$article = Article::find(1);
// 增加一个关联数据
$article->comments()->save(['content'=>'test']);
// 批量增加关联数据
$article->comments()->saveAll([
    ['content'=>'thinkphp'],
    ['content'=>'onethink'],
]);
```

定义相对的关联

要在 Comment 模型定义相对应的关联，可使用 belongsTo 方法：

```
name app\index\model;

use think\Model;

class Comment extends Model
{
    public function article()
    {
        return $this->belongsTo('article');
    }
}
```

远程一对多

远程一对多关联用于定义有跨表的一对多关系，例如：

- 每个城市有多个用户
- 每个用户有多个话题
- 城市和话题之间并无关联

关联定义

就可以直接通过远程一对多关联获取每个城市的多个话题，`City` 模型定义如下：

```
namespace app\index\model;

use think\Model;

class City extends Model
{
    public function topics()
    {
        return $this->hasManyThrough('Topic', 'User');
    }
}
```

远程一对多关联，需要同时存在 `Topic` 和 `User` 模型。

`hasManyThrough` 方法的参数如下：

`hasManyThrough('关联模型名','中间模型名','外键名','中间模型关联键名','当前模型主键名',['模型别名定义']);`

关联查询

我们可以通过下面的方式获取关联数据

```
$city = City::get(1);
// 获取同城的所有话题
dump($city->topics);
// 也可以进行条件搜索
dump($city->topics()->where('topic.status',1)->select());
```

条件搜索的时候，需要带上模型名作为前缀

多对多关联

多对多关联 关联定义

例如，我们的用户和角色就是一种多对多的关系，我们在User模型定义如下：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany('Role');
    }
}
```

belongsToMany方法的参数如下：

belongsToMany('关联模型名','中间表名','外键名','当前模型关联键名',['模型别名定义']);

关联查询

我们可以通过下面的方式获取关联数据

```
$user = User::get(1);
// 获取用户的所有角色
dump($user->roles);
// 也可以进行条件搜索
dump($user->roles()->where('name','admin')->select());
```

关联新增

```
$user = User::get(1);
// 增加关联数据 会自动写入中间表数据
$user->roles()->save(['name'=>'管理员']);
// 批量增加关联数据
$user->roles()->saveAll([
    ['name'=>'管理员'],
    ['name'=>'操作员'],
]);
```

只新增中间表数据，可以使用

```
$user = User::get(1);
// 仅增加关联的中间表数据
$user->roles()->save(1);
// 或者
```

```
$role = Role::get(1);
$user->roles()->save($role);
// 批量增加关联数据
$user->roles()->saveAll([1,2,3]);
```

单独更新中间表数据，可以使用：

```
$user = User::get(1);
// 增加关联的中间表数据
$user->roles()->attach(1);
// 传入中间表的额外属性
$user->roles()->attach(1,['remark'=>'test']);
// 删除中间表数据
$user->roles()->detach([1,2,3]);
```

定义相对的关联

我们可以在 `Role` 模型中定义一个相对的关联关系，例如：

```
namespace app\index\model;

use think\Model;

class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany('User');
    }
}
```

动态属性

模型对象的关联属性可以直接作为当前模型对象的动态属性进行赋值或者取值操作，虽然该属性并非数据表字段，例如：

```
namespace app\index\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile');
    }
}
```

我们在使用

```
// 查询模型数据
$user = User::find(1);
// 获取动态属性
dump($user->profile);
// 给关联模型属性赋值
$user->profile->phone = '1234567890';
// 保存关联模型数据
$user->profile->save();
```

在获取动态属性 `profile` 的同时，模型会通过定义的关联方法去查询关联对象的数据并赋值给该动态属性，这是一种关联数据的“惰性加载”，只有真正访问关联属性的时候才会进行关联查询。

当有大量的关联数据需要查询的时候，一般都会考虑选择关联预载入的方式（参考下一节）。

关联预载入

关联查询的预查询载入功能，主要解决了 **N+1** 次查询的问题，例如下面的查询如果有3个记录，会执行4次查询：

```
$list = User::all([1,2,3]);
foreach($list as $user){
    // 获取用户关联的profile模型数据
    dump($user->profile);
}
```

如果使用关联预查询功能，对于一对一关联来说，只有一次查询，对于一对多关联的话，就可以变成2次查询，有效提高性能。

```
$list = User::with('profile')->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的profile模型数据
    dump($user->profile);
}
```

支持预载入多个关联，例如：

```
$list = User::with('profile,book')->select([1,2,3]);
```

也可以支持嵌套预载入，例如：

```
$list = User::with('profile.phone')->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的phone模型
    dump($user->profile->phone);
}
```

可以在模型的get和all方法中使用预载入，和使用select方法是等效的：

```
$list = User::all([1,2,3], 'profile,book');
```

如果要指定属性查询，可以使用：

```
$list = User::field('id,name')->with(['profile'=>function($query){$query->withField('email,phone');}])->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的profile模型数据
    dump($user->profile);
}
```


聚合模型

通过聚合模型可以把**一对一关联**的操作更加简化，只需要把你的模型类继承 `think\model\Merge`，就可以自动完成关联查询、关联保存和关联删除。

例如下面的用户表关联了档案表，两个表信息如下：

think_user

| 字段名 | 描述 |
|----------|-----|
| id | 主键 |
| name | 用户名 |
| password | 密码 |
| nickname | 昵称 |

think_profile

| 字段名 | 描述 |
|----------|------|
| id | 主键 |
| truename | 真实姓名 |
| phone | 电话 |
| email | 邮箱 |
| user_id | 用户ID |

我们只需要定义好主表的模型，例如下面是User模型的定义：

```
namespace app\index\model;

use think\model\Merge;

class User extends Merge
{
    // 定义关联模型列表
    protected static $relationModel = ['Profile'];
    // 定义关联外键
    protected $fk = 'user_id';
    protected $mapFields = [
        // 为混淆字段定义映射
        'id' => 'User.id',
        'profile_id' => 'Profile.id',
    ];
}
```

如果需要单独设置关联数据表，可以使用：

```
namespace app\index\model;

use think\model\Merge;
```

```
class User extends Merge
{
    // 设置主表名
    protected $table = 'think_user';
    // 定义关联模型列表
    protected static $relationModel = [
        // 给关联模型设置数据表
        'Profile' => 'think_user_profile',
    ];
    // 定义关联外键
    protected $fk = 'user_id';
    protected $mapFields = [
        // 为混淆字段定义映射
        'id' => 'User.id',
        'profile_id' => 'Profile.id',
    ];
}
```

注意：对于关联表中存在混淆的字段名一定要通过mapFields属性定义。

接下来，我们可以和使用普通模型一样的方法来操作用户模型及其关联数据。

```
// 关联查询
$user = User::get(1);
echo $user->id;
echo $user->name;
echo $user->phone;
echo $user->email;
echo $user->profile_id;
$user->email = 'thinkphp@qq.com';
// 关联保存
$user->save();
// 关联删除
$user->delete();
// 根据主键关联删除
User::destroy([1,2,3]);
```

操作两个数据表就和操作一个表一样的感觉，关联表的写入、更新和删除自动采用事务（只要数据库支持事务），一旦主表写入失败或者发生异常就会发生回滚。

如果主表除了Profile关联之外，还有其他的一对多关联，一样可以定义额外的关联，例如：

```
namespace app\index\model;
use think\model\Merge;

class User extends Merge
{
    // 定义关联模型列表
    protected static $relationModel = ['Profile'];
    // 定义关联外键
    protected $fk = 'user_id';
    protected $mapFields = [
        // 为混淆字段定义映射
        'id' => 'User.id',
    ];
}
```

```
        'profile_id' => 'Profile.id',  
    ];  
  
    public function articles(){  
        return $this->hasMany('Article');  
    }  
}
```

对一对多关联进行操作，例如：

```
$user = User::get(1);  
// 读取关联信息  
dump($user->articles);  
// 或者进行关联预载入  
$user = User::get(1, 'articles');
```

注意：不能再次对 已经在 `relationModel` 属性中定义过的关联表进行关联定义和预载入查询。

视图

[视图实例化](#)

[模板引擎](#)

[模板赋值](#)

[模板渲染](#)

[输出替换](#)

视图实例化

视图功能由 `\think\View` 类配合视图驱动（模板引擎）类一起完成，目前的内置模板引擎包含PHP原生模板和Think模板引擎。

因为新版的控制器可以无需继承任何的基础类，因此在控制器中如何使用视图取决于你怎么定义控制器。

直接实例化视图类

任何情况下，你都可以直接实例化视图类进行渲染模板。

```
// 实例化视图类
$view = new View();
// 渲染模板输出 并赋值模板变量
return $view->fetch('hello', ['name'=>'thinkphp']);
```

实例化视图类的时候，可以传入模板引擎相关配置参数，例如：

```
// 实例化视图类
$view = new View([
    'type'          => 'think',
    'view_path'      => '',
    'view_suffix'    => 'html',
    'view_depr'      => '/',
]);
// 渲染模板输出 并赋值模板变量
return $view->fetch('hello', ['name'=>'thinkphp']);
```

如果需要使用应用自己扩展的模板引擎驱动，可以使用：

```
// 实例化视图类
$view = new View([
    'type'          => '\org\template\Think',
    'view_path'      => '',
    'view_suffix'    => 'html',
    'view_depr'      => '/',
]);
```

如果实例化不当，很容易导致配置参数无效的情况。因此如果不是必要，不建议直接实例化 `View` 类进行操作。

继承 `\think\Controller` 类

如果你的控制器继承了 `\think\Controller` 类的话，则无需自己实例化视图类，可以直接调用控制器基础类封装的相关视图类的方法。

```
// 渲染模板输出
```

```
return $this->fetch('hello',['name'=>'thinkphp']);
```

| 下面的方法可以直接被调用： | 方法 | 说明 |
|---------------|---------|----|
| fetch | 渲染模板输出 | |
| display | 渲染内容输出 | |
| assign | 模板变量赋值 | |
| engine | 初始化模板引擎 | |

如果需要调用View类的其它方法，可以直接使用 `$this->view` 对象：

助手函数

如果你只是需要渲染模板输出的话，可以使用系统提供的助手函数 `view`，可以完成相同的功能：

```
return view('hello',['name'=>'thinkphp']);
```

模板引擎

内置模板引擎

视图的模板文件可以支持不同的解析规则，默认情况下无需手动初始化模板引擎。

可以通过下面的几种方式对模板引擎进行初始化。

配置文件

在应用配置文件中配置 `template` 参数即可，例如：

```
'template'                => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'                  => 'Think',
    // 模板路径
    'view_path'             => './template/',
    // 模板后缀
    'view_suffix'           => 'html',
    // 模板文件名分隔符
    'view_depr'             => DS,
    // 模板引擎普通标签开始标记
    'tpl_begin'             => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'               => '}',
    // 标签库标签开始标记
    'taglib_begin'          => '{',
    // 标签库标签结束标记
    'taglib_end'            => '}',
],
```

调用视图类进行操作或者使用 `view` 助手函数的时候会自动实例化相关的模板引擎并传入参数。

实例化视图

可以在实例化视图的时候直接传入模板引擎配置参数，会在渲染输出的时候自动初始化模板引擎，例如：

```
$view = new View([
    'type'                  => 'think',
    'view_path'             => './template/',
    'view_suffix'           => 'php',
    'view_depr'             => DS,
    'tpl_begin'             => '{', // 模板引擎普通标签开始标记
    'tpl_end'               => '}', // 模板引擎普通标签结束标记
    'strip_space'           => true, // 去除模板文件里面的html空格与换行
    'tpl_cache'             => true, // 开启模板编译缓存
    'taglib_pre_load'       => '', // 需要额外加载的标签库(须指定标签库名称)，多个以逗号分隔
    'tpl_replace_string'    => [], // 模板过滤输出（与输出替换章节不同，前者对模版进行过滤）
]);
```

`think` 模板引擎是ThinkPHP内置的一个基于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部分。

调用engine方法初始化

视图类也提供了 `engine` 方法对模板解析引擎进行初始化或者切换不同的模板引擎，例如：

```
$view = new View();  
return $view->engine('php')->fetch();
```

表示当前视图的模板文件使用原生php进行解析。

设置模板引擎参数

除了在实例化的时候传入外，可以动态设置模板引擎的相关参数，例如：

```
$view = new View();  
return $view->config('view_path', './template/')->fetch();
```

使用第三方模板引擎

官方扩展库中提供了一个类似于 `angularjs` 语法的模板引擎 `think-angular`，具体可以参考[参考手册](#)。

模板赋值

模板赋值

除了系统变量和配置参数输出无需赋值外，其他变量如果需要在模板中输出必须首先进行模板赋值操作，绑定数据到模板输出有三种方式：

assign 方法

```
namespace index\app\controller;

class Index extends \think\Controller
{
    public function index()
    {
        // 模板变量赋值
        $this->assign('name', 'ThinkPHP');
        $this->assign('email', 'thinkphp@qq.com');
        // 或者批量赋值
        $this->assign([
            'name' => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
        // 模板输出
        return $this->fetch('index');
    }
}
```

传入参数方法

方法fetch 及 display 均可传入模版变量，例如

```
namespace app\index\controller;

class Index extends \think\Controller
{
    public function index()
    {
        return $this->fetch('index', [
            'name' => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
    }
}
```

```
class Index extends \think\Controller
{
    public function index()
    {
        $content = '{$name}-{$email}';
        return $this->display($content, [
            'name' => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
    }
}
```

```
}
```

对象赋值

```
class Index extends \think\Controller
{
    public function index()
    {
        $view = $this->view;
        $view->name      = 'ThinkPHP';
        $view->email      = 'thinkphp@qq.com';
        // 模板输出
        return $view->fetch('index');
    }
}
```

助手函数

如果使用view助手函数渲染输出的话，可以使用下面的方法进行模板变量赋值：

```
return view('index', [
    'name' => 'ThinkPHP',
    'email' => 'thinkphp@qq.com'
]);
```

模板渲染

模板渲染

渲染模板最常用的是使用 `\think\View` 类的 `fetch` 方法，调用格式：

```
fetch('模板文件','模板变量（数组）')
```

模板文件的写法支持下面几种：

| 用法 | 描述 |
|-----------------|----------------------|
| 不带任何参数 | 自动定位当前操作的模板文件 |
| [模块@][控制器/][操作] | 常用写法，支持跨模块 |
| 完整的模板文件名 | 直接使用完整的模板文件名（包括模板后缀） |

下面是一个最典型的用法，不带任何参数：

```
// 不带任何参数 自动定位当前操作的模板文件
$view = new View();
return $view->fetch();
```

表示系统会按照默认规则自动定位模板文件，其规则是：

```
当前模块/默认视图目录/当前控制器（小写）/当前操作（小写）.html
```

如果有更改模板引擎的 `view_depr` 设置（假设 `'view_depr'=>'_'`）的话，则上面的自动定位规则变成：

```
当前模块/默认视图目录/当前控制器（小写）_当前操作（小写）.html
```

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

```
// 指定模板输出
return $view->fetch('edit');
```

表示调用当前控制器下面的edit模板

```
return $view->fetch('member/read');
```

表示调用Member控制器下面的read模板。

跨模块渲染模板

```
return $view->fetch('admin@member/edit');
```

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

```
return $view->fetch('public/menu');
```

输出这个模板文件。理解了这个，模板输出就清晰了。

`fetch` 方法支持在渲染输出的时候传入模板变量，例如：

```
return $view->fetch('read', ['a'=>'a', 'b'=>'b']);
```

自定义模板路径

如果你的模板文件位置比较特殊或者需要自定义模板文件的位置，可以采用下面的几种方式处理。

渲染完整模板

```
return $view->fetch('./template/public/menu.html');
```

这种方式需要带模板路径和后缀指定一个完整的模板文件位置，这里的 `template/public` 目录是位于当前项目入口文件位置下面。如果是其他的后缀文件，也支持直接输出，例如：

```
return $view->fetch('./template/public/menu.tpl');
```

只要 `./template/public/menu.tpl` 是一个实际存在的模板文件。

要注意模板文件位置是相对于应用的入口文件，而不是模板目录。

渲染内容

如果希望直接解析内容而不通过模板文件的话，可以使用 `display` 方法：

```
$view = new View();
return $view->display($content,$vars);
```

渲染的内容中一样可以使用模板引擎的相关标签。

输出替换

模板输出替换

支持对视图输出的内容进行字符替换，例如：

```
namespace index\app\controller;

class Index extends \think\Controller
{
    public function index()
    {
        $this->assign('name', 'thinkphp');
        return $this->fetch('index', [], ['__PUBLIC__'=>'/public/']);
    }
}
```

如果需要全局替换的话，可以直接在配置文件中添加：

```
'view_replace_str' => [
    '__PUBLIC__'=>'/public/',
    '__ROOT__' => '/',
]
```

然后就可以直接使用

```
namespace index\app\controller;

class Index extends \think\Controller
{
    public function index()
    {
        $this->assign('name', 'thinkphp');
        return $this->fetch('index');
    }
}
```

如果你手动实例化视图类，请确保在实例化的时候传入配置参数：

```
$view = new View([], Config::get('view_replace_str'));
return $view->fetch();
```

助手函数 `view` 也支持全局配置参数 `view_replace_str` 的设置，如果需要设置不同的替换参数，可以使用：

```
return view('index', ['name'=>'thinkphp'], ['__PUBLIC__'=>'/public/']);
```

在渲染模板或者内容输出的时候就会自动根据设置的替换规则自动替换。

要使得你的全局替换生效，确保你的控制器类继承`think\Controller`或者使用`view`助手函数渲染输出。

模板

本章的内容主要讲述了如何使用内置的模板引擎来定义模板文件，以及使用加载文件、模板布局和模板继承等高级功能。

ThinkPHP内置了一个基于XML的性能卓越的模板引擎，这是一个专门为ThinkPHP服务的内置模板引擎，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。

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

| 标签类型 | 描述 |
|-------|--------------------------|
| 普通标签 | 主要用于输出变量和做一些基本的操作 |
| XML标签 | 主要完成一些逻辑判断、控制和循环输出，并且可扩展 |

这种方式的结合保证了模板引擎的简洁和强大的有效融合。

模板定位

模板文件定义

每个模块的模板文件是独立的，为了对模板文件更加有效的管理，ThinkPHP对模板文件进行目录划分，默认的模板文件定义规则是：

视图目录/控制器名（小写）/操作名（小写）+模板后缀

默认的视图目录是**模块的view目录**，框架的默认视图文件后缀是 `.html`。

模板渲染规则

模板渲染使用 `\think\View` 类的 `fetch` 方法，渲染规则为：

模块@控制器/操作

模板文件目录默认位于模块的view目录下面，视图类的fetch方法中的模板文件的定位规则如下：

如果调用没有任何参数的fetch方法：

```
return $view->fetch();
```

则按照系统的默认规则定位模板文件到：

[模板文件目录]/当前控制器名（小写+下划线）/当前操作名（小写）.html

如果（指定操作）调用：

```
return $view->fetch('add');
```

则定位模板文件为：

[模板文件目录]/当前控制器名（小写+下划线）/add.html

如果调用控制器的某个模板文件使用：

```
return $view->fetch('user/add');
```

则定位模板文件为：

[模板文件目录]/user/add.html

跨模块调用模板

```
return $view->fetch('admin@user/add');
```

全路径模板调用：

```
return $view->fetch(APP_PATH.request()->module().'/view/public/header.html');
```

模板标签

模板文件可以包含普通标签和标签库标签，标签的定界符都可以重新配置。

普通标签

普通标签用于变量输出和模板注释，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 }` 都属于正确的标签，而 `{ $name }`、`{ $vo.name }` 则不属于。

要更改普通标签的起始标签和结束标签，可以更改下面的配置参数：

```
'template' => [  
  // 模板引擎  
  'type'    => 'think',  
  // 普通标签开始标记  
  'tpl_begin' => '<{',  
  // 普通标签结束标记  
  'tpl_end'   => '}>',  
],
```

普通标签的定界符就被修改了，原来的 `{ $name }` 和 `{ $vo.name }` 必须使用 `<{ $name }>` 和 `<{ $vo.name }>` 才能生效了。

标签库标签

标签库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。

5.0版本的标签库默认定界符和普通标签一样使用 `{` 和 `}`，是为了便于在编辑器里面编辑不至于报错，当然，你仍然可以更改标签库标签的起始和结束标签，修改下面的配置参数：

```
'template'      => [  
  // 模板引擎  
  'type'         => 'think',  
  // 标签库标签开始标签  
  'taglib_begin' => '<',  
  // 标签库标签结束标记  
  'taglib_end'   => '>',  
],
```

原来的

```
{eq name="name" value="value"}  
相等  
{else/}  
不相等  
{/eq}
```

就需要改成

```
<eq name="name" value="value">
相等
<else/>
不相等
</eq>
```

变量输出

在模板中输出变量的方法很简单，例如，在控制器中我们给模板变量赋值：

```
$view = new View();  
$view->name = 'thinkphp';  
return $view->fetch();
```

然后就可以在模板中使用：

```
Hello, {$name} !
```

模板编译后的结果就是：

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

这样，运行的时候就会在模板中显示：**Hello, ThinkPHP !**

注意模板标签的 { 和 \$ 之间不能有任何的空格，否则标签无效。所以，下面的标签

```
Hello, { $name} !
```

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

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

```
$data['name'] = 'ThinkPHP';  
$data['email'] = 'thinkphp@qq.com';  
$view->assign('data', $data);
```

那么，在模板中我们可以用下面的方式输出：

```
Name : {$data.name}  
Email : {$data.email}
```

或者用下面的方式也是有效：

```
Name : {$data['name']}  
Email : {$data['email']}
```

当我们要输出多维数组的时候，往往要采用后面一种方式。

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

```
Name : {$data:name}  
Email : {$data:email}
```

或者

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

系统变量

系统变量输出

普通的模板变量需要首先赋值后才能在模板中输出，但是系统变量则不需要，可以直接在模板中输出，系统变量的输出通常以`{Think`打头，例如：

```
{Think.server.script_name} // 输出$_SERVER['SCRIPT_NAME']变量
{Think.session.user_id} // 输出$_SESSION['user_id']变量
{Think.get.pageNumber} // 输出$_GET['pageNumber']变量
{Think.cookie.name} // 输出$_COOKIE['name']变量
```

支持输出 `$_SERVER`、`$_ENV`、`$_POST`、`$_GET`、`$_REQUEST`、`$_SESSION` 和 `$_COOKIE` 变量。

常量输出

还可以输出常量

```
{Think.const.APP_PATH}
```

或者直接使用

```
{Think.APP_PATH}
```

配置输出

输出配置参数使用：

```
{Think.config.default_module}
{Think.config.default_controller}
```

语言变量

输出语言变量可以使用：

```
{Think.lang.page_error}
{Think.lang.var_error}
```

请求参数

模板支撑直接输出 `Request` 请求对象的方法参数，用法如下：

`$Request.方法名.参数`

例如：

```
{ $Request.get.id }  
{ $Request.param.name }
```

以 `$Request.` 开头的变量输出会认为是系统Request请求对象的参数输出。

支持 `Request` 类的大部分方法，但只支持方法的第一个参数。

下面都是有效的输出：

```
// 调用Request对象的get方法 传入参数为id  
{ $Request.get.id }  
// 调用Request对象的param方法 传入参数为name  
{ $Request.param.name }  
// 调用Request对象的param方法 传入参数为user.nickname  
{ $Request.param.user.nickname }  
// 调用Request对象的root方法  
{ $Request.root }  
// 调用Request对象的root方法，并且传入参数true  
{ $Request.root.true }  
// 调用Request对象的path方法  
{ $Request.path }  
// 调用Request对象的module方法  
{ $Request.module }  
// 调用Request对象的controller方法  
{ $Request.controller }  
// 调用Request对象的action方法  
{ $Request.action }  
// 调用Request对象的ext方法  
{ $Request.ext }  
// 调用Request对象的host方法  
{ $Request.host }  
// 调用Request对象的ip方法  
{ $Request.ip }  
// 调用Request对象的header方法  
{ $Request.header.accept-encoding }
```


使用函数

我们往往需要对模板输出变量使用函数，可以使用：

```
{ $data.name|md5 }
```

编译后的结果是：

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

如果函数有多个参数需要调用，则使用：

```
{ $create_time|date="y-m-d",### }
```

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

```
<?php echo (date("y-m-d",$create_time)); ?>
```

如果前面输出的变量在后面定义的函数的第一个参数，则可以直接使用：

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

表示输出

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

虽然也可以使用：

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

但完全没用这个必要。

还可以支持多个函数过滤，多个函数之间用“|”分割即可，例如：

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

编译后的结果是：

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

函数会按照从左到右的顺序依次调用。

如果你觉得这样写起来比较麻烦，也可以直接这样写：

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

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

使用默认值

我们可以给变量输出提供默认值，例如：

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

对系统变量依然可以支持默认值输出，例如：

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

默认值和函数可以同时使用，例如：

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

使用运算符

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

例如：

| 运算符 | 使用示例 |
|------|-----------------------|
| + | { \$a+\$b } |
| - | { \$a-\$b } |
| * | { \$a*\$b } |
| / | { \$a/\$b } |
| % | { \$a%\$b } |
| ++ | { \$a++ } 或 { ++\$a } |
| -- | { \$a-- } 或 { --\$a } |
| 综合运算 | { \$a+\$b*10+\$c } |

在使用运算符的时候，不再支持常规函数用法，例如：

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

三元运算

模板可以支持三元运算符，例如：

```
{ $status? '正常' : '错误' }  
{ $info['status']? $info['msg'] : $info['error'] }  
{ $info.status? $info.msg : $info.error }
```

5.0版本还支持如下的写法：

```
{ $varname.aa ?? 'xxx' }
```

表示如果有设置\$varname则输出\$varname,否则输出'xxx'。解析后的代码为：

```
<?php echo isset($varname['aa']) ? $varname['aa'] : '默认值'; ?>
```

```
{ $varname?='xxx' }
```

表示\$varname为真时才输出xxx。解析后的代码为：

```
<?php if(!empty($name)) echo 'xxx'; ?>
```

```
{ $varname ?: 'no' }
```

表示如果\$varname为真则输出\$varname，否则输出no。解析后的代码为：

```
<?php echo $varname ? $varname : 'no'; ?>
```

```
{ $a==$b ? 'yes' : 'no' }
```

前面的表达式为真输出yes,否则输出no，条件可以是==、===、!=、!==、>=、<=

原样输出

可以使用 `literal` 标签来防止模板标签被解析，例如：

```
{literal}
  Hello, {$name} !
{/literal}
```

上面的 `{$name}` 标签被 `literal` 标签包含，因此并不会被模板引擎解析，而是保持原样输出。

`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。

需要注意的是配置 `'view_replace_str'` 替换参数，会替换掉 `literal` 标签内的内容，可以配置 `'template.tpl_replace_string'` 避免替换掉 `literal` 标签内的内容。

模板注释

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

单行注释

格式：

```
{/* 注释内容 */ } 或 {// 注释内容 }
```

例如：

```
{// 这是模板注释内容 }
```

注意 { 和注释标记之间不能有空格。

多行注释

支持多行注释，例如：

```
{/* 这是模板  
注释内容*/ }
```

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

模板布局

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

有三种布局模板的支持方式：

第一种方式：全局配置方式

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

```
'template' => [  
    'layout_on'    => true,  
    'layout_name' => 'layout',  
]
```

开启 `layout_on` 后，我们的模板渲染流程就有所变化，例如：

```
namespace app\index\controller;  
  
use think\Controller;  
  
class User extends Controller  
{  
    public function add()  
    {  
        return $this->fetch('add');  
    }  
}
```

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

```
{include file="public/header" /}  
{__CONTENT__}  
{include file="public/footer" /}
```

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

当然可以通过设置来改变这个特定的替换字符串，例如：

```
'template' => [  
    'layout_on'    => true,  
    'layout_name' => 'layout',  
    'layout_content' => 'CONTENT',  
]
```



```
'layout_on'      => true,
'layout_name'    => 'layout',
'layout_item'    => '{__REPLACE__}'
]
```

一个布局模板同时只能有一个特定替换字符串。

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

如果需要指定其他位置的布局模板，可以使用：

```
'template' => [
  'layout_on'      => true,
  'layout_name'    => 'layout/layoutname',
  'layout_item'    => '{__REPLACE__}'
]
```

就表示采用 application/index/view/layout/layoutname.html 作为布局模板。

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

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

第二种方式：模板标签方式

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

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

```
{layout name="layout" /}
```

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

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

如果需要使用其他的布局模板，可以改变layout的name属性，例如：

```
{layout name="newlayout" /}
```

还可以在layout标签里面指定要替换的特定字符串：

```
{layout name="Layout/newlayout" replace="[__REPLACE__]" /}
```

第三种方式：使用layout控制模板布局

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

```
namespace app\index\controller;

use think\Controller;

class User extends Controller
{
    public function add()
    {
        $this->view->engine->layout(true);
        return $this->fetch('add');
    }
}
```

表示当前的模板输出启用了布局模板，并且采用默认的layout布局模板。

如果当前输出需要使用不同的布局模板，可以动态的指定布局模板名称，例如：

```
namespace app\index\controller;

use think\Controller;

class User extends Controller
{
    public function add()
    {
        $this->view->engine->layout('Layout/newlayout');
        return $this->display('add');
    }
}
```

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

```
namespace app\index\controller;

use think\Controller;

class User extends Controller
{
    public function add()
    {
        // 临时关闭当前模板的布局功能
        $this->view->engine->layout(false);
        return $this->display('add');
    }
}
```

```
}  
}
```

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

模板继承

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

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

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

```
{block name="title"><title>网站标题</title>{/block}
```

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

```
{block name="title"><title>{$web_title}</title>{/block}
```

你甚至还可以在区块中加载外部文件：

```
{block name="include"}{include file="Public:header" /}{/block}
```

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

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>{block name="title"}标题{/block}</title>
</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}{/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="main"}
{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"}
    {__block__}
    @ThinkPHP 版权所有
{/block}

```

上例中，我们可以看到在子模板中使用了extend标签来继承了base模板。

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。而

```

{block name="footer"}
    {__block__}@ThinkPHP 版权所有
{/block}

```

这一区块中有{__block__}这个标签，当区块中有这个标记时，就不只是直接重载这个区块，它表示引用所继承模板对应区块的内容到这个位置，最终这个区块是合并后的内容。所以这里footer区块最后的内容是：底部 @ThinkPHP 版权所有

extend标签的用法和include标签一样，你也可以加载其他模板：

```
{extend name="Public:base" /}
```

或者使用绝对文件路径加载

```
{extend name="./Template/Public/base.html" /}
```

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

例如，如果采用下面的定义：

```

{block name="title"><title>{$title}</title>{/block}
<a href="/" >首页</a>

```

```
<a href="/info/" >资讯</a>  
<a href="/bbs/" >论坛</a>
```

导航部分将是无效的，不会显示在模板中。

模板可以多级继承，比如B继承了A，而C又继承了B，最终C中的区块会覆盖B和A中的同名区块，但C和B中的区块必须是A中已定义过的。

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。

包含文件

在当前模版文件中包含其他的模版文件使用include标签，标签用法：

```
{include file='模版文件1,模版文件2,...' /}
```

包含的模板文件中不能再使用模板布局或者模板继承。

使用模版表达式

模版表达式的定义规则为：**模块@控制器/操作**

例如：

```
{include file="public/header" /} // 包含头部模版header
{include file="public/menu" /} // 包含菜单模版menu
{include file="blue/public/menu" /} // 包含blue主题下面的menu模版
```

可以一次包含多个模版，例如：

```
{include file="public/header,public/menu" /}
```

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

使用模版文件

可以直接包含一个模版文件名（包含完整路径），例如：

```
{include file="../../../application/view/default/public/header.html" /}
```

路径以 项目目录/public/ 路径下为起点

传入参数

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

```
{include file="Public/header" title="$title" keywords="开源WEB开发框架" /}
```

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

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

上面title参数传入的是个变量\$title，模板内的[title]最终会替换成\$title的值，当然\$title这个变量必须要存在。

包含文件中可以再使用include标签包含别的文件，但注意不要形成A包含A，或者A包含B而B又包含A这样的死循环。

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。

标签库

内置的模板引擎除了支持普通变量的输出之外，更强大的地方在于标签库功能。

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

标签库分为内置和扩展标签库，内置标签库是 **Cx** 标签库。

导入标签库

使用taglib标签导入当前模板中需要使用的标签库，例如：

```
{taglib name="html" /}
```

如果没有定义html标签库的话，则导入无效。

也可以导入多个标签库，使用：

```
{taglib name="html,article" /}
```

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

```
{article:read name="hello" id="data" }  
{$data.id}:{$data.title}  
{/article:read}
```

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

如果是 `{article:read name="hello" /}` 就是开放标签。

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

内置标签

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

```
{eq name="status" value="1" }  
正常  
{/eq}
```

如果Cx不是内置标签的话，可能就需要这么使用了：

```
{cx:eq name="status" value="1" }  
正常  
{/cx:eq}
```

更多的Cx标签库中的标签用法，参考[内置标签](#)。

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

```
'taglib_build_in'    =>    'cx,article'
```

配置后，上面的标签用法就可以改为：

```
{read name="hello" id="data" }  
{$data.id}:{$data.title}  
{/read}
```

标签库预加载

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

在应用或者模块的配置文件中添加：

```
'taglib_pre_load'    =>    'article,html'
```

设置后，模板文件就不再需要使用

```
{taglib name="html,article" /}
```

但是仍然可以在模板中调用：

```
{article:read name="hello" id="data" }  
{$data.id}:{$data.title}  
{/article:read}
```

内置标签

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

内置标签包括：

| 标签名 | 作用 | 包含属性 |
|------------|--|-------------------------------|
| include | 包含外部模板文件（ 闭合 ） | file |
| load | 导入资源文件（ 闭合 包括js css import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 |

循环输出标签

VOLIST标签

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

```
$list = User::all();  
$this->assign('list',$list);
```

在模版定义如下，循环输出用户的编号和姓名：

```
{volist name="list" id="vo"}  
{$vo.id}:{$vo.name}<br/>  
{/volist}
```

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

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

支持输出查询结果中的部分数据，例如输出其中的第5 ~ 15条记录

```
{volist name="list" id="vo" offset="5" length='10'}  
{$vo.name}  
{/volist}
```

输出偶数记录

```
{volist name="list" id="vo" mod="2" }  
{eq name="mod" value="1"}{$vo.name}{/eq}  
{/volist}
```

Mod属性还用于控制一定记录的换行，例如：

```
{volist name="list" id="vo" mod="5" }  
{$vo.name}  
{eq name="mod" value="4"}<br/>{/eq}  
{/volist}
```

为空的时候输出提示：

```
{volist name="list" id="vo" empty="暂时没有数据" }
{$vo.id}|{$vo.name}
{/volist}
```

empty属性不支持直接传入html语法，但可以支持变量输出，例如：

```
$this->assign('empty','<span class="empty">没有数据</span>');
$this->assign('list',$list);
```

然后在模板中使用：

```
{volist name="list" id="vo" empty="$empty" }
{$vo.id}|{$vo.name}
{/volist}
```

输出循环变量：

```
{volist name="list" id="vo" key="k" }
{$k}.{$vo.name}
{/volist}
```

如果没有指定key属性的话，默认使用循环变量i，例如：

```
{volist name="list" id="vo" }
{$i}.{$vo.name}
{/volist}
```

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

```
{volist name="list" id="vo" }
{$key}.{$vo.name}
{/volist}
```

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

```
{volist name=":fun('arg')" id="vo"}
{$vo.name}
{/volist}
```

FOREACH标签

foreach标签类似与volist标签，只是更加简单，没有太多额外的属性，最简单的用法是：

```
{foreach $list as $vo}
{$vo.id}:{$vo.name}
```

```
{/foreach}
```

该用法解析后是最简洁的。

也可以使用下面的用法：

```
{foreach name="list" item="vo"}
    {$vo.id}:{$vo.name}
{/foreach}
```

name表示数据源 item表示循环变量。

可以输出索引，如下：

```
{foreach name="list" item="vo" }
    {$key}|{$vo}
{/foreach}
```

也可以定义索引的变量名

```
{foreach name="list" item="vo" key="k" }
    {$k}|{$vo}
{/foreach}
```

FOR标签

用法：

```
{for start="开始值" end="结束值" comparison="" step="步进值" name="循环变量名" }
{/for}
```

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

```
{for start="1" end="100"}
{$i}
{/for}
```

解析后的代码是

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

比较标签

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

```
{比较标签 name="变量" value="值"}
内容
{/比较标签}
```

系统支持的比较标签以及所表示的含义分别是：

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

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

例如，要求name变量的值等于value就输出，可以使用：

```
{eq name="name" value="value"}value{/eq}
```

或者

```
{equal name="name" value="value"}value{/equal}
```

也可以支持和else标签混合使用：

```
{eq name="name" value="value"}
相等
{else/}
不相等
{/eq}
```

当 name变量的值大于5就输出

```
{gt name="name" value="5"}value{/gt}
```

当name变量的值不小于5就输出

```
{egt name="name" value="5"}value{/egt}
```

比较标签中的变量可以支持对象的属性或者数组，甚至可以是系统变量，例如：当vo对象的属性（或者数组，或者自动判断）等于5就输出

```
{eq name="vo.name" value="5"}
{$vo.name}
{/eq}
```

当vo对象的属性等于5就输出

```
{eq name="vo:name" value="5"}
{$vo.name}
{/eq}
```

当\$vo['name']等于5就输出

```
{eq name="vo['name']" value="5"}
{$vo.name}
{/eq}
```

而且还可以支持对变量使用函数 当vo对象的属性值的字符串长度等于5就输出

```
{eq name="vo:name|strlen" value="5"}{$vo.name}{/eq}
```

变量名可以支持系统变量的方式，例如：

```
{eq name="Think.get.name" value="value"}相等{/eq}{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

条件判断

SWITCH标签

用法：

```
{switch name="变量" }
  {case value="值1" break="0或1"}输出内容1{/case}
  {case value="值2"}输出内容2{/case}
  {default /}默认情况
{/switch}
```

使用方法如下：

```
{switch name="User.level"}
  {case value="1"}value1{/case}
  {case value="2"}value2{/case}
  {default /}default
{/switch}
```

其中name属性可以使用函数以及系统变量，例如：

```
{switch name="Think.get.userId|abs"}
  {case value="1"}admin{/case}
  {default /}default
{/switch}
```

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

```
{switch name="Think.get.type"}
  {case value="gif|png|jpg"}图像格式{/case}
  {default /}其他格式
{/switch}
```

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

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

```
{switch name="Think.get.userId|abs"}
  {case value="1" break="0"}admin{/case}
  {case value="2"}admin{/case}
  {default /}default
{/switch}
```

也可以对case的value属性使用变量，例如：

```
{switch name="User.userId"}
  {case value="$adminId"}admin{/case}
  {case value="$memberId"}member{/case}
  {default /}default
{/switch}
```

使用变量方式的情况下，不再支持多个条件的同时判断。

简洁的用法

```
{switch $User.userId}
  {case $adminId}admin{/case}
  {case $memberId}member{/case}
{/switch}
```

IF标签

用法示例：

```
{if condition="($name == 1) OR ($name > 100) "} value1
{elseif condition="$name eq 2"/}value2
{else /} value3
{/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。

简洁的用法

```
{if condition="表达式"}  
{if (表达式)}  
{if 表达式}
```

这三种写法结果是一样的

范围判断

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

IN和NOTIN

用法： 假设我们中控制器中给id赋值为1：

```
$id = 1;  
$this->assign('id',$id);
```

我们可以使用in标签来判断模板变量是否在某个范围内，例如：

```
{in name="id" value="1,2,3"}  
id在范围内  
{/in}
```

最后会输出：**id在范围内**。

如果判断不在某个范围内，可以使用notin标签：

```
{notin name="id" value="1,2,3"}  
id不在范围内  
{/notin}
```

最后会输出：**id不在范围内**。

可以把上面两个标签合并成为：

```
{in name="id" value="1,2,3"}  
id在范围内  
{else/}  
id不在范围内  
{/in}
```

name属性还可以支持直接判断系统变量，例如：

```
{in name="Think.get.id" value="1,2,3"}
```

```
$_GET['id'] 在范围内  
{/in}
```

更多的系统变量用法可以参考[系统变量](#)部分。

value属性也可以使用变量，例如：

```
{in name="id" value="$range"}  
id在范围内  
{/in}
```

\$range变量可以是数组，也可以是以逗号分隔的字符串。

value属性还可以使用系统变量，例如：

```
{in name="id" value="$Think.post.ids"}  
id在范围内  
{/in}
```

BETWEEN 和 NOTBETWEEN

可以使用between标签来判断变量是否在某个区间范围内，可以使用：

```
{between name="id" value="1,10"}  
输出内容1  
{/between}
```

同样，也可以使用notbetween标签来判断变量不在某个范围内：

```
{notbetween name="id" value="1,10"}  
输出内容2  
{/notbetween}
```

也可以使用else标签把两个用法合并，例如：

```
{between name="id" value="1,10"}  
输出内容1  
{else/}  
输出内容2  
{/between}
```

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

```
{between name="id" value="1,3,10"}  
输出内容1  
{/between}
```

实际判断的范围区间是 `1~3`，而不是 `1~10`，也可以支持字符串判断，例如：

```
{between name="id" value="A,Z"}  
输出内容1  
{/between}
```

name属性可以直接使用系统变量，例如：

```
{between name="Think.post.id" value="1,5"}  
输出内容1  
{/between}
```

value属性也可以使用变量，例如：

```
{between name="id" value="$range"}  
输出内容1  
{/between}
```

变量的值可以是字符串或者数组，还可以支持系统变量。

```
{between name="id" value="$Think.get.range"}  
输出内容1  
{/between}
```

RANGE

也可以直接使用range标签，替换前面的判断用法：

```
{range name="id" value="1,2,3" type="in"}  
输出内容1  
{/range}
```

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

PRESENT NOTPRESENT标签

present标签用于判断某个变量是否已经定义，用法：

```
{present name="name"}  
name已经赋值  
{/present}
```

如果判断没有赋值，可以使用：

```
{notpresent name="name"}  
name还没有赋值  
{/notpresent}
```

可以把上面两个标签合并成为：

```
{present name="name"}  
name已经赋值  
{else /}  
name还没有赋值  
{/present}
```

present标签的name属性可以直接使用系统变量，例如：

```
{present name="Think.get.name"}  
$_GET['name']已经赋值  
{/present}
```

EMPTY NOTEMPTY 标签

empty标签用于判断某个变量是否为空，用法：

```
{empty name="name"}  
name为空值  
{/empty}
```

如果判断没有赋值，可以使用：

```
{notempty name="name"}  
name不为空  
{/notempty}
```

可以把上面两个标签合并成为：

```
{empty name="name"}  
name为空  
{else /}  
name不为空  
{/empty}
```

name属性可以直接使用系统变量，例如：

```
{empty name="Think.get.name"}  
$_GET['name']为空值  
{/empty}
```

DEFINED 标签

DEFINED标签用于判断某个常量是否有定义，用法如下：

```
{defined name="NAME"}  
NAME常量已经定义
```

```
{/defined}
```

name属性的值要注意严格大小写

如果判断没有被定义，可以使用：

```
{notdefined name="NAME"}  
NAME常量未定义  
{/notdefined}
```

可以把上面两个标签合并成为：

```
{defined name="NAME"}  
NAME常量已经定义  
{else /}  
NAME常量未定义  
{/defined}
```


资源文件加载

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='/static/js/common.js'>
<link rel="stylesheet" type="text/css" href="/static/css/style.css" />
```

系统提供了专门的标签来简化上面的导入：

```
{load href="/static/js/common.js" /}
{load href="/static/css/style.css" /}
```

并且支持同时加载多个资源文件，例如：

```
{load href="/static/js/common.js,/static/css/style.css" /}
```

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="/static/js/common.js" /}
{css href="/static/css/style.css" /}
```

标签嵌套

模板引擎支持标签的多层嵌套功能，可以对标签库的标签指定可以嵌套。

系统内置的标签中，`volist`、`switch`、`if`、`elseif`、`else`、`foreach`、`compare`（包括所有的比较标签）、`(not) present`、`(not) empty`、`(not) defined`等标签都可以嵌套使用。例如：

```
{volist name="list" id="vo"}
    {volist name="vo['sub']" id="sub"}
        {$sub.name}
    {/volist}
{/volist}
```

上面的标签可以用于输出双重循环。

原生PHP

Php代码可以和标签在模板文件中混合使用，可以在模板文件里面书写任意的PHP语句代码，包括下面两种方式：

使用php标签

例如：

```
{php}echo 'Hello,world!';{/php}
```

我们建议需要使用PHP代码的时候尽量采用php标签，因为原生的PHP语法可能会被配置禁用而导致解析错误。

使用原生php代码

```
<?php echo 'Hello,world!'; ?>
```

注意：php标签或者php代码里面就不能再使用标签（包括普通标签和XML标签）了，因此下面的几种方式都是无效的：

```
{php}{eq name='name' value='value'}value{/eq}{/php}
```

Php标签里面使用了 `eq` 标签，因此无效

```
{php}if( {$user} != 'ThinkPHP' ) echo 'ThinkPHP' ;{/php}
```

Php标签里面使用了 `{$user}` 普通标签输出变量，因此无效。

```
{php}if( $user.name != 'ThinkPHP' ) echo 'ThinkPHP' ;{/php}
```

Php标签里面使用了 `$user.name` 点语法变量输出，因此无效。

简而言之，在PHP标签里面不能再使用PHP本身不支持的代码。

如果设置了 `tpl_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。

定义标签

ASSIGN标签

ASSIGN标签用于在模板文件中定义变量，用法如下：

```
{assign name="var" value="123" /}
```

在运行模板的时候，赋值了一个 `var` 的变量，值是 `123`。

name属性支持系统变量，例如：

```
{assign name="Think.get.id" value="123" /}
```

表示在模板中给 `$_GET['id']` 赋值了 `123`

value属性也支持变量，例如：

```
{assign name="var" value="$val" /}
```

或者直接把系统变量赋值给var变量，例如：

```
{assign name="var" value="$Think.get.name" /}
```

相当于，执行了：`$var = $_GET['name'];`

DEFINE标签

DEFINE标签用于中模板中定义常量，用法如下：

```
{define name="MY_DEFINE_NAME" value="3" /}
```

在运行模板的时候，就会定义一个 `MY_DEFINE_NAME` 的常量。

value属性可以支持变量（包括系统变量），例如：

```
{define name="MY_DEFINE_NAME" value="$name" /}
```

或者

```
{define name="MY_DEFINE_NAME" value="$Think.get.name" /}
```


日志

介绍

日志记录由 `\think\Log` 类完成，主要完成日志记录和跟踪调试。由于日志记录了所有的运行错误，因此养成经常查看日志文件的习惯，可以避免和及早发现很多的错误隐患。

日志初始化

在使用日志记录之前，首先需要初始化日志类，指定当前使用的日志记录方式。

```
Log::init([
    'type' => 'File',
    'path' => APP_PATH.'logs/'
]);
```

上面在日志初始化的时候，指定了文件方式记录日志，并且日志保存目录为 `APP_PATH.'logs/'`。

如果你没有执行日志初始化操作的话，默认会自动调用配置参数 `log` 来进行初始化。

不同的日志类型可能会使用不同的初始化参数。

如果应用需要扩展自己的日志驱动，可以使用：

```
Log::init([
    'type' => '\org\Log\File',
    'path' => APP_PATH.'logs/'
]);
```

日志驱动

日志驱动

日志可以通过驱动支持不同的方式写入，默认日志会记录到文件中，系统已经内置的写入驱动包括 **File**、**Socket**，如果要临时关闭日志写入，可以设置日志类型为Test即可，例如：

```
'log' => [  
    // 可以临时关闭日志写入  
    'type' => 'test',  
],
```

File 驱动

日志的记录方式默认是 **File** 方式，可以通过驱动的方式来扩展支持更多的记录方式。

记录方式由 **log.type** 参数配置，例如：

```
'log' => [  
    // 日志记录方式，支持 file socket  
    'type' => 'File',  
    //日志保存目录  
    'path' => LOG_PATH,  
    //单个日志文件的大小限制，超过后会自动记录到第二个文件  
    'file_size' => 2097152,  
    //日志的时间格式，默认是`c`  
    'time_format' => 'c'  
],
```

为了避免同一个目录下面的日志文件过多的性能问题，**file** 方式记录的日志文件会自动生成日期子目录。

Socket 驱动

Socket驱动配置，具体参考后面的 **远程调试** 部分。

其他驱动

thinkphp5.0支持 **SAE** 驱动的扩展，具体参考“SAE”章节

每个日志记录方式需要对应一个日志驱动文件，例如File方式记录，对应的驱动文件是 `library/think/log/driver/File.php`。

日志写入

手动记录

一般情况下，系统的日志记录是自动的，无需手动记录，但是某些时候也需要手动记录日志信息，Log类提供了3个方法用于记录日志。

| 方法 | 描述 |
|---------------|--------------------------|
| Log::record() | 记录日志信息到内存 |
| Log::save() | 把保存在内存中的日志信息（用指定的记录方式）写入 |
| Log::write() | 实时写入一条日志信息 |

由于系统在请求结束后会自动调用Log::save方法，所以通常，你只需要调用Log::record记录日志信息即可。

record方法用法如下：

```
Log::record('测试日志信息');
```

默认的话记录的日志级别是INFO，也可以指定日志级别：

```
Log::record('测试日志信息，这是警告级别','notice');
```

采用record方法记录的日志信息不是实时保存的，如果需要实时记录的话，可以采用write方法，例如：

```
Log::write('测试日志信息，这是警告级别，并且实时写入','notice');
```

日志级别

ThinkPHP对系统的日志按照级别来分类，并且这个日志级别完全可以自己定义，系统内部使用的级别包括：

- **log** 常规日志，用于记录日志
- **error** 错误，一般会导致程序的终止
- **notice** 警告，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，用于SQL记录，只在数据库的调试模式开启时有效

系统提供了不同日志级别的快速记录方法，例如：

```
Log::error('错误信息');
Log::info('日志信息');
// 和下面的用法等效
Log::record('错误信息','error');
Log::record('日志信息','info');
```

还封装了一个助手函数用于日志记录，例如：

```
trace('错误信息','error');
trace('日志信息','info');
```

也支持指定级别日志的输入，需要配置信息：

```
'log'    => [
  'type'  => 'File',
  // 日志记录级别，使用数组表示
  'level' => ['error'],
],
```

独立日志

为了便于分析，**File** 类型的日志驱动还支持设置某些级别的日志信息单独文件记录，例如：

```
'log' => [  
  'type' => 'file',  
  // error和sql日志单独记录  
  'apart_level' => ['error', 'sql'],  
],
```

设置后，就会单独生成 **error** 和 **sql** 两个类型的日志文件，主日志文件中将不再包含这两个级别的日志信息。

独立日志文件按天保存，不限制大小。

日志清空

日志类提供了日志清空的方法，可以在需要的时候手动清空日志，日志清空仅仅是清空内存中的日志。

使用方法如下：

```
Log::clear();
```

写入授权

5.0的日志功能支持写入授权，我们可以设置某个请求的日志授权Key，然后设置允许授权写入的配置 **Key**，实现个别用户日志记录的功能，从而提高高负载下面的日志记录性能。

首先需要在应用配置文件或者应用公共文件中添加当前访问的授权Key定义，例如：

```
// 设置IP为授权Key
Log::key(Request::instance()->ip());
```

然后在日志配置参数中增加 **allow_key** 参数，如下：

```
'log' => [
    // 日志类型为File
    'type' => 'File',
    // 授权只有202.12.36.89 才能记录日志
    'allow_key' => ['202.12.36.89'],
]
```

错误和调试

[调试模式](#)

[异常处理](#)

[抛出异常](#)

[Trace调试](#)

[变量调试](#)

[性能调试](#)

[SQL调试](#)

[远程调试](#)

[404页面](#)

调试模式

ThinkPHP有专门为开发过程而设置的调试模式，开启调试模式后，会牺牲一定的执行效率，但带来的方便和除错功能非常值得。

我们强烈建议ThinkPHP开发人员在开发阶段始终开启调试模式（直到正式部署后关闭调试模式），方便及时发现隐患问题和分析、解决问题。

应用默认开启调试模式，在完成开发阶段部署到生产环境后，可以修改应用配置文件的 `app_debug` 参数关闭调试模式切换到部署模式。

```
// 关闭调试模式
'app_debug' => false,
```

除此之外，还可以在应用的 `ROOT_PATH` 目录下面定义 `.env` 文件，并且定义 `APP_DEBUG` 配置参数用于替代入口文件的常量定义，这样便于在部署环境中设置环境变量来开启和关闭调试模式。

`.env` 文件的定义格式如下：

```
// 设置开启调试模式
app_debug = true
// 其它的环境变量设置
// ...
```

定义了 `.env` 文件后，配置文件中定义 `app_debug` 参数无效。

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；
- 会详细记录整个执行过程；
- 模板修改可以即时生效；
- 记录SQL日志，方便分析SQL；
- 通过Trace功能更好的调试和发现错误；
- 发生异常的时候会显示详细的异常信息；

由于调试模式没有任何缓存，因此涉及到较多的文件IO操作和模板实时编译，所以在开启调试模式的情况下，性能会有一定的下降，但不会影响部署模式的性能。另外需要注意的是，一旦关闭调试模式，项目的调试配置文件即刻失效。

一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以

如下设置：

```
// 显示错误信息  
'show_error_msg'      =>  true,
```


异常处理

和PHP默认的异常处理不同，ThinkPHP抛出的不是单纯的错误信息，而是一个人性化的错误页面。

默认异常处理

在调试模式下，系统默认展示的错误页面：

[0] [HttpException](#) in App.php line 313

模块不存在:hello

```
304.         $available = true;
305.     }
306.
307.     // 模块初始化
308.     if ($module && $available) {
309.         // 初始化模块
310.         $request->module($module);
311.         $config = self::init($module);
312.     } else {
313.         throw new HttpException(404, 'module not exists:' . $module);
314.     }
315. } else {
316.     // 单一模块部署
317.     $module = '';
318.     $request->module($module);
319. }
320. // 当前模块路径
321. App::$modulePath = APP_PATH . ($module ? $module . DS : '');
322.
```

只有在调试模式下面才能显示具体的错误信息，如果在部署模式下面，你可能看到的是一个简单的提示文字，例如：

页面错误！请稍后再试~

ThinkPHP V5.0.0 RC4 { 十年磨一剑-为API开发设计的高性能框架 }

本着严谨的原则，5.0版本默认情况下会对任何错误（包括警告错误）抛出异常，如果不希望如此严谨的

抛出异常，可以在应用公共函数文件中或者配置文件中使用 `error_reporting` 方法设置错误报错级别（请注意，在入口文件中设置是无效的），例如：

```
// 异常错误报错级别，
error_reporting(E_ERROR | E_PARSE );
```

异常处理接管

框架支持异常页面由开发者自定义类进行处理，需要配置参数 `exception_handle`

```
// 异常处理handle类 留空使用 \think\exception\Handle
'exception_handle' => '\\app\common\exception\Http',
```

自定义类需要继承 `Handle` 并且实现 `render` 方法，可以参考如下代码：

```
<?php
namespace app\common\exception;
use think\exception\Handle;
use think\exception\HttpException;
class Http extends Handle
{
    public function render(\Exception $e)
    {
        if ($e instanceof HttpException) {
            $statusCode = $e->getStatusCode();
        }
        //TODO::开发者对异常的操作
        //可以在此交由系统处理
        return parent::render($e);
    }
}
```

需要注意的是，如果配置了 `'exception_handle'`，且没有再次调用系统 `render` 的情况下，配置 `http_exception_template` 就不再生效，具体可以参考 `Handle` 类内实现的功能。

部署模式异常

一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以如下设置：

```
// 显示错误信息
'show_error_msg' => true,
```

模块不存在:hello

ThinkPHP V5.0.0 RC4 { 十年磨一剑-为API开发设计的高性能框架 }

异常捕获

可以使用PHP的异常捕获进行必要的处理，但需要注意一点，在异常捕获中不要使用 `think\Controller` 类的 `error`、`success` 和 `redirect` 方法，因为上述三个方法会抛出 `HttpResponseException` 异常，从而影响正常的异常捕获，例如：

```
try{
    Db::name('user')->find();
    $this->success('执行成功!');
}catch(\Exception $e){
    $this->error('执行错误');
}
```

应该改成

```
try{
    Db::name('user')->find();
}catch(\Exception $e){
    $this->error('执行错误');
}
$this->success('执行成功!');
```

抛出异常

手动抛出异常

可以使用 `\think\Exception` 类来抛出异常

```
// 使用think自带异常类抛出异常
throw new \think\Exception('异常消息', 1000006);
```

如果不使用think异常类，也可以定义自己的异常类来抛出异常

```
throw new \foobar\Exception('异常消息');
```

也可以使用系统提供的助手函数来简化处理：

```
exception('异常消息', 1000006);

// 使用自定义异常类
exception('异常消息', 1000006, \foobar\Exception);
```

抛出 HTTP 异常

可以使用 `\think\exception\HttpException` 类来抛出异常

```
// 抛出 HTTP 异常
throw new \think\exception\HttpException(404, '异常消息', null, [参数]);
```

系统提供了助手函数 `abort` 简化HTTP异常的处理，例如：

```
abort(404, '异常消息', [参数])
```

HTTP异常可以单独定义异常模板，请参考后面的404页面。

Trace调试

调试模式并不能完全满足我们调试的需要，有时候我们需要手动的输出一些调试信息。除了本身可以借助一些开发工具进行调试外，ThinkPHP还提供了一些内置的调试工具和函数。例如，**Trace** 调试功能就是ThinkPHP提供给开发人员的一个用于开发调试的辅助工具。可以实时显示当前页面的操作的请求信息、运行情况、SQL执行、错误提示等，并支持自定义显示，5.0版本的Trace调试支持没有页面输出的操作调试。

Trace调试功能对调试模式和部署模式都有效，可以单独开启和关闭。

只是在部署模式下面，显示的调试信息没有调试模式完整，通常我们建议Trace配合调试模式一起使用。

开启Trace调试

默认关闭Trace调试功能，要开启Trace调试功能，只需要配置下面参数：

```
// 开启应用Trace调试
'app_trace' => true,
```

如果定义了环境变量 `app_trace`，那么以环境变量配置为准。

页面Trace显示

要开启页面Trace功能，需要配置 `trace` 参数为：

```
// Trace信息
'trace' => [
    //支持Html,Console
    'type' => 'html',
]
```

设置后并且你的页面有输出的话，页面右下角会显示 **ThinkPHP** 的LOGO：



我们看到的LOGO后面的数字就是当前页面的执行时间（单位是秒）点击该图标后，会展开详细的Trace信息，如图：

基本 文件 流程 错误 SQL 调试

1. 请求信息：2016-03-12 14:03:39 HTTP/1.1 GET : localhost/tp5/public/
2. 运行时间：0.003126s [吞吐率：319.90req/s] 内存消耗：58.30kb 文件加载：28
3. 查询信息：0 queries 0 writes
4. 缓存信息：0 reads,0 writes
5. 配置加载：55

Trace框架有6个选项卡，分别是基本、文件、流程、错误、SQL和调试，点击不同的选项卡会切换到不同的Trace信息窗口。

| 选项卡 | 描述 |
|-----|--------------------------------------|
| 基本 | 当前页面的基本摘要信息，例如执行时间、内存开销、文件加载数、查询次数等等 |
| 文件 | 详细列出当前页面执行过程中加载的文件及其大小 |
| 流程 | 会列出当前页面执行到的行为和相关流程 |
| 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 |

Trace的选项卡是可以定制和扩展的，默认的配置为：

```
// 显示Trace信息
'trace' =>[
  'type'      =>  'Html',
  'trace_tabs' => [
    'base'=>'基本',
    'file'=>'文件',
    'info'=>'流程',
    'error|notice'=>'错误',
    'sql'=>'SQL',
    'debug|log'=>'调试'
  ]
]
```

也就是我们看到的默认情况下显示的选项卡，如果你希望增加新的选项卡：用户，则可以修改配置如下：

```
// 显示Trace信息
'trace' =>[
  'type'      =>  'Html',
  'trace_tabs' => [
    'base'=>'基本',
    'file'=>'文件',
    'info'=>'流程',
    'error'=>'错误',
    'sql'=>'SQL',
    'debug'=>'调试',
    'user'=>'用户'
  ]
]
```

也可以把某几个选项卡合并，例如：

```
// 显示Trace信息
'trace' =>[
  'type'      =>  'Html',
  'trace_tabs' => [
    'base'=>'基本',
    'file'=>'文件',
    'error|notice'=>'错误',
```

```

    'sql'=>'SQL',
    'debug|log|info'=>'调试',
  ]
]

```

更改后的Trace显示效果如图：

| 基本 | 文件 | 错误 | SQL | 调试 |
|--|----|----|-----|----|
| 1. 请求信息：2016-03-12 14:15:39 HTTP/1.1 GET : localhost/tp5/public/ | | | | |
| 2. 运行时间：0.002965s [吞吐率：337.27req/s] 内存消耗：58.05kb 文件加载：28 | | | | |
| 3. 查询信息：0 queries 0 writes | | | | |
| 4. 缓存信息：0 reads,0 writes | | | | |
| 5. 配置加载：55 | | | | |

浏览器Trace显示

trace功能支持在浏览器的 `console` 直接输出，这样可以方便没有页面输出的操作功能调试，只需要设置：

```

// Trace信息
'trace' =>[
  // 使用浏览器console输出trace信息
  'type' => 'console',
]

```

运行后打开浏览器的console控制台可以看到如图所示的信息：

| |
|---|
| ▼ 基本 |
| 请求信息 2016-06-29 21:51:56 HTTP/1.1 GET : localhost/tp/public/ |
| 运行时间 0.026364088058472s [吞吐率：37.93req/s] 内存消耗：79.52kb 文件加载：30 |
| 查询信息 0 queries 0 writes |
| 缓存信息 0 reads,0 writes |
| 配置加载 56 |
| ► 文件 |
| ► 流程 |
| ▼ 错误 |
| ► sql |
| ▼ 调试 |

浏览器Trace输出仍然支持 `trace_tabs` 设置。

变量调试

除了Trace调试之外，系统还提供了 `\think\Debug` 类用于各种调试。

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

用法：

```
Debug::dump($var, $echo=true, $label=null)
或者
dump($var, $echo=true, $label=null)
```

相关参数的使用如下：

| 参数 | 描述 |
|--------------|--------------------------------|
| var (必须) | 要输出的变量，支持所有变量类型 |
| echo (可选) | 是否直接输出，默认为true，如果为false则返回但不输出 |
| label (可选) | 变量输出的label标识，默认为空 |

如果echo参数为false 则返回要输出的字符串

使用示例：

```
$blog = Db::name('blog')->where('id', 3)->find();

Debug::dump($blog);
// 下面的用法是等效的
dump($blog);
```

在浏览器输出的结果是：

```
array(12) {
    ["id"] => string(1) "3"
    ["name"] => string(0) ""
    ["user_id"] => string(1) "0"
    ["cate_id"] => string(1) "0"
    ["title"] => string(4) "test"
    ["content"] => string(4) "test"
    ["create_time"] => string(1) "0"
    ["update_time"] => string(1) "0"
    ["status"] => string(1) "0"
    ["read_count"] => string(1) "0"
    ["comment_count"] => string(1) "0"
    ["tags"] => string(0) ""
}
```


如果需要在调试变量输出后中止程序的执行，可以使用 `halt` 函数，例如：

```
$blog = Db::name('blog')->where('id', 3)->find();  
  
halt($blog);  
echo '这里的信息是看不到的';
```

执行后会输出

```
array(12) {  
    ["id"] => string(1) "3"  
    ["name"] => string(0) ""  
    ["user_id"] => string(1) "0"  
    ["cate_id"] => string(1) "0"  
    ["title"] => string(4) "test"  
    ["content"] => string(4) "test"  
    ["create_time"] => string(1) "0"  
    ["update_time"] => string(1) "0"  
    ["status"] => string(1) "0"  
    ["read_count"] => string(1) "0"  
    ["comment_count"] => string(1) "0"  
    ["tags"] => string(0) ""  
}
```

并中止执行后续的程序。

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

`think\Debug` 类可以很方便的获取某个区间的运行时间和内存占用情况。 例如：

```
Debug::remark('begin');
// ...其他代码段
Debug::remark('end');
// ...也许这里还有其他代码
// 进行统计区间
echo Debug::getRangeTime('begin','end').'s';
```

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

默认的统计精度是小数点后4位，如果觉得这个统计精度不够，还可以设置例如：

```
echo Debug::getRangeTime('begin','end',6).'s';
```

可能的输出会变成：`0.005587s`

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

```
echo Debug::getRangeMem('begin','end').'kb';
```

第三个参数使用m表示进行内存开销统计，输出的结果可能是：`625kb`

同样，如果end标签没有被标记的话，会自动把当前位置先标记位end标签。

助手函数

系统还提供了助手函数 `debug` 用于完成相同的作用，上面的代码可以改成：

```
debug('begin');
// ...其他代码段
debug('end');
// ...也许这里还有其他代码
// 进行统计区间
echo debug('begin','end').'s';
echo debug('begin','end',6).'s';
echo debug('begin','end','m').'kb';
```

SQL调试

查看SQL记录

如果开启了数据库的调试模式的话，可以在日志文件（或者设置的日志输出类型）中看到详细的SQL执行记录以及性能分析。

下面是一个典型的SQL日志：

```
[ SQL ] SHOW COLUMNS FROM `think_action` [ RunTime:0.001339s ]
[ EXPLAIN : array ( 'id' => '1', 'select_type' => 'SIMPLE', 'table' => 'think_action',
'partitions' => NULL, 'type' => 'ALL', 'possible_keys' => NULL, 'key' => NULL, 'key_len'
' => NULL, 'ref' => NULL, 'rows' => '82', 'filtered' => '100.00', 'extra' => NULL, ) ]
[ SQL ] SELECT * FROM `think_action` LIMIT 1 [ RunTime:0.000539s ]
```

监听SQL

如果开启数据库的调试模式的话，你可以对数据库执行的任何SQL操作进行监听，使用如下方法：

```
Db::listen(function($sql,$time,$explain){
    // 记录SQL
    echo $sql. ' ['. $time. 's]';
    // 查看性能分析结果
    dump($explain);
});
```

默认如果没有注册任何监听操作的话，这些SQL执行会被根据不同的日志类型记录到日志中。

调试执行的SQL语句

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

```
User::get(1);
echo User::getLastSql();
```

输出结果是 `SELECT * FROM 'think_user' WHERE 'id' = '1'`

也可以使用 `fetchSql` 方法直接返回当前的查询SQL而不执行，例如：

```
echo User::fetchSql()->find(1);
```

输出的结果是一样的。

`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 Trace 信息或者日志文件。

远程调试

ThinkPHP5.0 版本开始，提供了 `Socket` 日志驱动用于本地和远程调试。

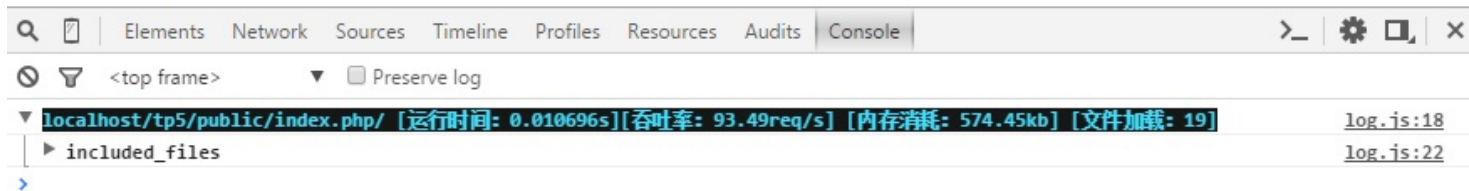
Socket调试

只需要在配置文件中设置如下：

```
'log' => [
    'type'           => 'socket',
    'host'           => 'slog.thinkphp.cn',
    //日志强制记录到配置的client_id
    'force_client_ids' => [],
    //限制允许读取日志的client_id
    'allow_client_ids' => [],
]
```

上面的host配置地址是官方提供的公用服务端，首先需要去[申请client_id](#)。

使用Chrome浏览器运行后，打开 `审查元素->Console`，可以看到如下所示：



SocketLog 通过 `websocket` 将调试日志打印到浏览器的 `console` 中。你还可以用它来分析开源程序，分析SQL性能，结合taint分析程序漏洞。

安装Chrome插件

SocketLog 首先需要安装 `chrome` 插件，Chrome[插件安装页面](#)（需翻墙）

使用方法

- 首先，请在chrome浏览器上安装好插件。
- 安装服务端 `npm install -g socketlog-server`，运行命令 `socketlog-server` 即可启动服务。将会在本地起一个websocket服务，监听端口是1229。
- 如果想服务后台运行：`socketlog-server > /dev/null &`

参数

- `client_id`：在chrome浏览器中，可以设置插件的 `Client_ID`，**Client_ID**是你任意指定的字符串。



- 设置 `client_id` 后能实现以下功能：
- 1, 配置 `allow_client_ids` 配置项，让指定的浏览器才能获得日志，这样就可以把调试代码带上线。普通用户访问不会触发调试，不会发送日志。开发人员访问就能看的调试日志，这样利于找线上bug。**Client_ID** 建议设置为姓名拼音加上随机字符串，这样如果有员工离职可以将其对应的 `client_id` 从配置项 `allow_client_ids` 中移除。`client_id` 除了姓名拼音，加上随机字符串的目的，以防别人根据你公司员工姓名猜测出 `client_id`，获取线上的调试日志。
- 设置 `allow_client_ids` 示例代码：

```
'allow_client_ids'=>['thinkphp_zfH5NbLn','luofei_DJq0z80H'],
```

- 2, 设置 `force_client_ids` 配置项，让后台脚本也能输出日志到chrome。网站有可能用了队列，一些业务逻辑通过后台脚本处理，如果后台脚本需要调试，你也可以将日志打印到浏览器的console中，当然后台脚本不和浏览器接触，不知道当前触发程序的是哪个浏览器，所以我们需要强制将日志打印到指定 `client_id` 的浏览器上面。我们在后台脚本中使用SocketLog时设置 `force_client_ids` 配置项指定要强制输出浏览器的 `client_id` 即可。

404页面

一旦抛出了 `HttpException` 异常，可以支持定义单独的异常页面的模板地址，只需要在应用配置文件中增加：

```
'http_exception_template' => [  
  // 定义404错误的重定向页面地址  
  404 => APP_PATH.'404.html',  
  // 还可以定义其它的HTTP status  
  401 => APP_PATH.'401.html',  
]
```

模板文件支持模板引擎中的标签。

`http_exception_template` 配置仅在部署模式下面生效。

一般来说 `HTTP` 异常是由系统自动抛出的，但我们也可以手动抛出

```
throw new \think\exception\HttpException(404, '页面不存在');
```

或者通过助手函数 `abort` 手动抛出 `HTTP` 异常，例如：

```
abort(404, '页面不存在');
```

验证

验证器

验证规则

错误信息

验证场景

控制器验证

模型验证

内置规则

静态调用

表单令牌

验证器

概述

ThinkPHP 5.0 验证使用独立的 `\think\Validate` 类或者验证器进行验证。

独立验证

任何时候，都可以使用 `Validate` 类进行独立的验证操作，例如：

```
$validate = new Validate([
    'name' => 'require|max:25',
    'email' => 'email'
]);
$data = [
    'name' => 'thinkphp',
    'email' => 'thinkphp@qq.com'
];
if (!$validate->check($data)) {
    dump($validate->getError());
}
```

验证器

这是 5.0 推荐的验证方式，为具体的验证场景或者数据表定义好验证器类，直接调用验证类的 `check` 方法即可完成验证，下面是一个例子：

我们定义一个 `\app\index\validate\User` 验证器类用于 `User` 的验证。

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name' => 'require|max:25',
        'email' => 'email',
    ];
}
```

在需要进行 `User` 验证的地方，添加如下代码即可：

```
$data = [
    'name'=>'thinkphp',
    'email'=>'thinkphp@qq.com'
];

$validate = Loader::validate('User');

if(!$validate->check($data)){
    dump($validate->getError());
}
```

```
}
```

使用助手函数实例化验证器

```
$validate = validate('User');
```

验证规则

设置规则

可以在实例化 `Validate` 类的时候传入验证规则，例如：

```
$rules = [  
    'name' => 'require|max:25',  
    'age'  => 'number|between:1,120',  
];  
$validate = new Validate($rules);
```

也可以使用 `rule` 方法动态添加规则，例如：

```
$rules = [  
    'name' => 'require|max:25',  
    'age'  => 'number|between:1,120',  
];  
$validate = new Validate($rules);  
$validate->rule('zip', '/^\d{6}$');  
$validate->rule([  
    'email' => 'email',  
]);
```

规则定义

规则定义支持下面两种方式：

```
$rules = [  
    'name' => 'require|max:25',  
    'age'  => 'number|between:1,120',  
];  
$validate = new Validate($rules);
```

或者

```
$rules = [  
    'name' => ['require', 'max'=>25],  
    'age'  => ['number', 'between'=>'1,120'],  
];  
$validate = new Validate($rules);
```

对于一个字段可以设置多个验证规则，使用 `|` 分割。

属性定义

通常情况下，我们实际在定义验证类的时候，可以通过属性的方式直接定义验证规则等信息，例如：

```
namespace app\index\validate;
```

```

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name' => 'require|max:25',
        'age'  => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message = [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];
}

```

验证数据

下面是一个典型的验证数据的例子：

```

$rule = [
    'name' => 'require|max:25',
    'age'  => 'number|between:1,120',
    'email' => 'email',
];

$msg = [
    'name.require' => '名称必须',
    'name.max'     => '名称最多不能超过25个字符',
    'age.number'   => '年龄必须是数字',
    'age.between'  => '年龄只能在1-120之间',
    'email'        => '邮箱格式错误',
];

$data = [
    'name' => 'thinkphp',
    'age'  => 10,
    'email' => 'thinkphp@qq.com',
];

$validate = new Validate($rule, $msg);
$result    = $validate->check($data);

```

如果需要批量验证，可以使用：

```

$validate = new Validate($rule, $msg);
$result    = $validate->batch()->check($data);

```

批量验证如果验证不通过，返回的是一个错误信息的数组。

如果你定义了User验证器类的话，可以使用下面的验证代码：

```
$data = [
    'name' => 'thinkphp',
    'age'   => 10,
    'email' => 'thinkphp@qq.com',
];
$validate = Loader::validate('User');
if(!$validate->check($data)){
    dump($validate->getError());
}
```

闭包函数验证

可以对某个字段使用闭包验证，例如：

```
$validate = new \think\Validate([
    'name' => function($value,$rule) {
        return $rule==$value ? true : false;
    },
]);
```

自定义验证规则

系统内置了一些常用的规则，如果还不够用，可以自己扩展验证规则。

如果使用了验证器的话，可以直接在验证器类添加自己的验证方法，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name' => 'checkName:thinkphp',
        'email' => 'email',
    ];

    protected $message = [
        'name' => '用户名必须',
        'email' => '邮箱格式错误',
    ];

    // 自定义验证规则
    protected function checkName($value,$rule,$data)
    {
        return $rule == $value ? true : '名称错误';
    }
}
```

验证方法支持传入的参数包括 \$value \$rule 和 \$data三个，并且第三个参数支持引用传参。

并且需要注意的是，自定义的验证规则方法名不能和已有的规则冲突。

接下来，就可以这样进行验证：

```
$validate = Loader::validate('User');
if(!$validate->check($data)){
    dump($validate->getError());
}
```

如果没有使用验证器类，则支持使用 **extend** 方法扩展验证规则，例如：

```
$validate = new Validate(['name' => 'checkName:1']);
$validate->extend('checkName', function ($value, $rule) {
    return $rule == $value ? true : '名称错误';
});
$data    = ['name' => 1];
$result = $validate->check($data);
```

支持批量注册验证规则，例如：

```
$validate = new Validate(['name' => 'checkName:1']);
$validate->extend([
    'checkName'=> function ($value, $rule) {
        return $rule == $value ? true : '名称错误';
    },
    'checkStatus'=> [$this, 'checkStatus']
]);
$data    = ['name' => 1];
$result = $validate->check($data);
```

错误信息

验证规则的错误提示信息有三种方式可以定义，如下：

使用默认的错误提示信息

如果没有定义任何的验证提示信息，系统会显示默认的错误信息，例如：

```
$rule = [  
    'name' => 'require|max:25',  
    'age' => 'number|between:1,120',  
    'email' => 'email',  
];  
  
$data = [  
    'name' => 'thinkphp',  
    'age' => 121,  
    'email' => 'thinkphp@qq.com',  
];  
$validate = new Validate($rule);  
$result = $validate->check($data);  
if(!$result){  
    echo $validate->getError();  
}
```

会输出 `age只能在 1 - 120 之间`。

可以给age字段设置中文名，例如：

```
$rule = [  
    'name' => 'require|max:25',  
    'age|年龄' => 'number|between:1,120',  
    'email' => 'email',  
];  
  
$data = [  
    'name' => 'thinkphp',  
    'age' => 121,  
    'email' => 'thinkphp@qq.com',  
];  
$validate = new Validate($rule);  
$result = $validate->check($data);  
if(!$result){  
    echo $validate->getError();  
}
```

会输出 `年龄只能在 1 - 120 之间`。

验证规则和提示信息分开定义

如果要输出自定义的错误信息，有两种方式可以设置。下面的一种方式是验证规则和提示信息分开定义：

```
$rule = [  

```

```

        'name' => 'require|max:25',
        'age'  => 'number|between:1,120',
        'email' => 'email',
    ];
    $msg = [
        'name.require' => '名称必须',
        'name.max'      => '名称最多不能超过25个字符',
        'age.number'    => '年龄必须是数字',
        'age.between'   => '年龄必须在1~120之间',
        'email'         => '邮箱格式错误',
    ];
    $data = [
        'name' => 'thinkphp',
        'age'  => 121,
        'email' => 'thinkphp@qq.com',
    ];
    $validate = new Validate($rule,$msg);
    $result    = $validate->check($data);
    if(!$result){
        echo $validate->getError();
    }
}

```

会输出 年龄必须在1~120之间。

验证规则和提示信息一起定义

可以支持验证规则和错误信息一起定义的方式，如下：

```

$rule = [
    ['name','require|max:25','名称必须|名称最多不能超过25个字符'],
    ['age','number|between:1,120','年龄必须是数字|年龄必须在1~120之间'],
    ['email','email','邮箱格式错误']
];

$data = [
    'name' => 'thinkphp',
    'age'  => 121,
    'email' => 'thinkphp@qq.com',
];
$validate = new Validate($rule);
$result    = $validate->check($data);
if(!$result){
    echo $validate->getError();
}

```


验证场景

可以在定义验证规则的时候定义场景，并且验证不同场景的数据，例如：

```
$rule = [
    'name' => 'require|max:25',
    'age' => 'number|between:1,120',
    'email' => 'email',
];
$msg = [
    'name.require' => '名称必须',
    'name.max' => '名称最多不能超过25个字符',
    'age.number' => '年龄必须是数字',
    'age.between' => '年龄只能在1-120之间',
    'email' => '邮箱格式错误',
];
$data = [
    'name' => 'thinkphp',
    'age' => 10,
    'email' => 'thinkphp@qq.com',
];
$validate = new Validate($rule);
$validate->scene('edit', ['name', 'age']);
$result = $validate->scene('edit')->check($data);
```

表示验证Edit场景（该场景定义只需要验证name和age字段）。

如果使用了验证器，可以直接在类里面定义场景，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name' => 'require|max:25',
        'age' => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message = [
        'name.require' => '名称必须',
        'name.max' => '名称最多不能超过25个字符',
        'age.number' => '年龄必须是数字',
        'age.between' => '年龄只能在1-120之间',
        'email' => '邮箱格式错误',
    ];

    protected $scene = [
        'edit' => ['name', 'age'],
    ];
}
```

可以在定义场景的时候对某些字段的规则重新设置，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name' => 'require|max:25',
        'age' => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message = [
        'name.require' => '名称必须',
        'name.max' => '名称最多不能超过25个字符',
        'age.number' => '年龄必须是数字',
        'age.between' => '年龄只能在1-120之间',
        'email' => '邮箱格式错误',
    ];

    protected $scene = [
        'edit' => ['name', 'age'=>'require|number|between:1,120'],
    ];
}
```

可以对场景设置一个回调方法，用于动态设置要验证的字段，例如：

```
$rule = [
    'name' => 'require|max:25',
    'age' => 'number|between:1,120',
    'email' => 'email',
];
$msg = [
    'name.require' => '名称必须',
    'name.max' => '名称最多不能超过25个字符',
    'age.number' => '年龄必须是数字',
    'age.between' => '年龄只能在1-120之间',
    'email' => '邮箱格式错误',
];
$data = [
    'name' => 'thinkphp',
    'age' => 10,
    'email' => 'thinkphp@qq.com',
];
$validate = new Validate($rule);
$validate->scene('edit', function($key,$data){
    return 'email'==$key && isset($data['id'])? true : false;
});
$result = $validate->scene('edit')->check($data);
```

控制器验证

如果你需要在控制器中进行验证，并且继承了 `\think\Controller` 的话，可以调用控制器类提供的 `validate` 方法进行验证，如下：

```
$result = $this->validate(  
    [  
        'name' => 'thinkphp',  
        'email' => 'thinkphp@qq.com',  
    ],  
    [  
        'name' => 'require|max:25',  
        'email' => 'email',  
    ]  
);  
if(true !== $result){  
    // 验证失败 输出错误信息  
    dump($result);  
}
```

如果定义了验证器类的话，例如：

```
namespace app\index\validate;  
  
use think\Validate;  
  
class User extends Validate  
{  
    protected $rule = [  
        'name' => 'require|max:25',  
        'email' => 'email',  
    ];  
  
    protected $message = [  
        'name.require' => '用户名必须',  
        'email' => '邮箱格式错误',  
    ];  
  
    protected $scene = [  
        'add' => ['name', 'email'],  
        'edit' => ['email'],  
    ];  
}
```

控制器中的验证代码可以简化为：

```
$result = $this->validate($data, 'User');  
if(true !== $result){  
    // 验证失败 输出错误信息  
    dump($result);  
}
```

如果要使用场景，可以使用：

本文档使用 [看云](#) 构建

```
$result = $this->validate($data, 'User.edit');  
if(true !== $result){  
    // 验证失败 输出错误信息  
    dump($result);  
}
```

在validate方法中还支持做一些前置的操作回调，使用方式如下：

```
$result = $this->validate($data, 'User.edit', [], [$this, 'some']);  
if(true !== $result){  
    // 验证失败 输出错误信息  
    dump($result);  
}
```

模型验证

在模型中的验证方式如下：

```
$User = new User;
$result = $User->validate(
    [
        'name' => 'require|max:25',
        'email' => 'email',
    ],
    [
        'name.require' => '名称必须',
        'name.max'      => '名称最多不能超过25个字符',
        'email'         => '邮箱格式错误',
    ]
)->save($data);
if(false === $result){
    // 验证失败 输出错误信息
    dump($User->getError());
}
```

第二个参数如果不传的话，则采用默认的错误提示信息。

如果使用下面的验证器类的话：

```
namespace app\index\validate;
use think\Validate;
class User extends Validate
{
    protected $rule = [
        'name' => 'require|max:25',
        'email' => 'email',
    ];

    protected $message = [
        'name.require' => '用户名必须',
        'email' => '邮箱格式错误',
    ];

    protected $scene = [
        'add' => ['name', 'email'],
        'edit' => ['email'],
    ];
}
```

模型验证代码可以简化为：

```
$User = new User;
// 调用当前模型对应的User验证器类进行数据验证
$result = $User->validate(true)->save($data);
if(false === $result){
    // 验证失败 输出错误信息
}
```

```
        dump($User->getError());  
    }
```

如果需要调用的验证器类和当前的模型名称不一致，则可以使用：

```
$User = new User;  
// 调用Member验证器类进行数据验证  
$result = $User->validate('Member')->save($data);  
if(false === $result){  
    // 验证失败 输出错误信息  
    dump($User->getError());  
}
```

同样也可以支持场景验证：

```
$User = new User;  
// 调用Member验证器类进行数据验证  
$result = $User->validate('User.edit')->save($data);  
if(false === $result){  
    // 验证失败 输出错误信息  
    dump($User->getError());  
}
```

内置规则

系统内置的验证规则如下：

格式验证类

require

验证某个字段必须，例如：

```
'name'=>'require'
```

number 或者 integer

验证某个字段的值是否为数字（采用 `filter_var` 验证），例如：

```
'num'=>'number'
```

float

验证某个字段的值是否为浮点数字（采用 `filter_var` 验证），例如：

```
'num'=>'float'
```

boolean

验证某个字段的值是否为布尔值（采用 `filter_var` 验证），例如：

```
'num'=>'boolean'
```

email

验证某个字段的值是否为email地址（采用 `filter_var` 验证），例如：

```
'email'=>'email'
```

array

验证某个字段的值是否为数组，例如：

```
'info'=>'array'
```

accepted

验证某个字段是否为为 yes, on, 或是 1。这在确认"服务条款"是否同意时很有用，例如：

```
'accept'=>'accepted'
```

date

验证值是否为有效的日期，例如：

```
'date'=>'date'
```

会对日期值进行 `strtotime` 后进行判断。

alpha

验证某个字段的值是否为字母，例如：

```
'name'=>'alpha'
```

alphaNum

验证某个字段的值是否为字母和数字，例如：

```
'name'=>'alphaNum'
```

alphaDash

验证某个字段的值是否为字母和数字，下划线 `_` 及破折号 `-`，例如：

```
'name'=>'alphaDash'
```

activeUrl

验证某个字段的值是否为有效的域名或者IP，例如：

```
'host'=>'activeUrl'
```


url

验证某个字段的值是否为有效的URL地址（采用 `filter_var` 验证），例如：

```
'url'=>'url'
```

ip

验证某个字段的值是否为有效的IP地址（采用 `filter_var` 验证），例如：

```
'ip'=>'ip'
```

支持验证ipv4和ipv6格式的IP地址。

dateFormat:format

验证某个字段的值是否为指定格式的日期，例如：

```
'create_time'=>'dateFormat:y-m-d'
```

长度和区间验证类

in

验证某个字段的值是否在某个范围，例如：

```
'num'=>'in:1,2,3'
```

notIn

验证某个字段的值不在某个范围，例如：

```
'num'=>'notIn:1,2,3'
```

between

验证某个字段的值是否在某个区间，例如：

```
'num'=>'between:1,10'
```

notBetween

验证某个字段的值不在某个范围，例如：

```
'num'=>'notBetween:1,10'
```

length:num1,num2

验证某个字段的值的长度是否在某个范围，例如：

```
'name'=>'length:4,25'
```

或者指定长度

```
'name'=>'length:4'
```

如果验证的数据是数组，则判断数组的长度。
如果验证的数据是File对象，则判断文件的大小。

max:number

验证某个字段的值的最大长度，例如：

```
'name'=>'max:25'
```

如果验证的数据是数组，则判断数组的长度。
如果验证的数据是File对象，则判断文件的大小。

min:number

验证某个字段的值的最小长度，例如：

```
'name'=>'min:5'
```

如果验证的数据是数组，则判断数组的长度。
如果验证的数据是File对象，则判断文件的大小。

after:日期

验证某个字段的值是否在某个日期之后，例如：

```
'begin_time' => 'after:2016-3-18',
```

before:日期

验证某个字段的值是否在某个日期之前，例如：

```
'end_time'    => 'before:2016-10-01',
```

expire:开始时间,结束时间

验证当前操作（注意不是某个值）是否在某个有效日期之内，例如：

```
'expire_time'  => 'expire:2016-2-1,2016-10-01',
```

allowIp:allow1,allow2,...

验证当前请求的IP是否在某个范围，例如：

```
'name'        => 'allowIp:114.45.4.55',
```

该规则可以用于某个后台的访问权限

denyIp:allow1,allow2,...

验证当前请求的IP是否禁止访问，例如：

```
'name'        => 'denyIp:114.45.4.55',
```

字段比较类

confirm

验证某个字段是否和另外一个字段的值一致，例如：

```
'repassport'=>'require|confirm:passport'
```

different

验证某个字段是否和另外一个字段的值不一致，例如：

```
'name'=>'require|different:account'
```

egt 或者 >=

验证是否大于等于某个值，例如：

```
'score'=>'egt:60'  
'num'=>'>=:100'
```

gt 或者 >

验证是否大于某个值，例如：

```
'score'=>'gt:60'  
'num'=>'>:100'
```

elt 或者 <=

验证是否小于等于某个值，例如：

```
'score'=>'elt:100'  
'num'=>'<=:100'
```

lt 或者 <

验证是否小于某个值，例如：

```
'score'=>'lt:100'  
'num'=>'<:100'
```

eq 或者 = 或者 same

验证是否等于某个值，例如：

```
'score'=>'eq:100'  
'num'=>'=:100'  
'num'=>'same:100'
```

filter验证

支持使用filter_var进行验证，例如：

```
'ip'=>'filter:validate_ip'
```

正则验证

支持直接使用正则验证，例如：

```
'zip'=>'d{6}',
// 或者
'zip'=>'regex:d{6}',
```

如果你的正则表达式中包含有 | 符号的话，必须使用数组方式定义。

```
'accepted'=>[ 'regex'=>'/(yes|on|1)$/i'],
```

也可以实现预定义正则表达式后直接调用，例如：

上传验证

file

验证是否是一个上传文件

image:width,height,type

验证是否是一个图像文件，width height和type都是可选，width和height必须同时定义。

fileExt:允许的文件后缀

验证上传文件后缀

fileMime:允许的文件类型

验证上传文件类型

fileSize:允许的文件字节大小

验证上传文件大小

行为验证

使用行为验证数据，例如：

```
'data'=>'behavior:\app\index\behavior\Check'
```

其它验证

unique:table,field,except,pk

验证当前请求的字段值是否为唯一的，例如：

```
// 表示验证name字段的值是否在user表（不包含前缀）中唯一
'name' => 'unique:user',
// 验证其他字段
```

```
'name'    => 'unique:user,account',  
// 排除某个主键值  
'name'    => 'unique:user,account,10',  
// 指定某个主键值排除  
'name'    => 'unique:user,account,10,user_id',
```

如果需要对复杂的条件验证唯一，可以使用下面的方式：

```
// 多个字段验证唯一验证条件  
'name'    => 'unique:user,status^account',  
// 复杂验证条件  
'name'    => 'unique:user,status=1&account= '.$data['account'],
```

requireIf:field,value

验证某个字段的值等于某个值的时候必须，例如：

```
// 当account的值等于1的时候 password必须  
'password'=>'requireIf:account,1'
```

requireWith:field

验证某个字段有值的时候必须，例如：

```
// 当account有值的时候password字段必须  
'password'=>'requireWith:account'
```

静态调用

如果需要使用内置的规则验证单个数据，可以使用静态调用的方式。

```
// 日期格式验证
Validate::dateFormat('2016-03-09','Y-m-d'); // true
// 验证是否有效的日期
Validate::is('2016-06-03','date'); // true
// 验证是否有效邮箱地址
Validate::is('thinkphp@qq.com','email'); // true
// 验证是否在某个范围
Validate::in('a',['a','b','c']); // true
// 验证是否大于某个值
Validate::gt(10,8); // true
// 正则验证
Validate::regex(100,'\d+'); // true
```

静态验证的返回值为布尔值，错误信息需要自己处理。

更多验证规则可以参考前面的内置规则。

表单令牌

验证规则支持对表单的令牌验证，首先需要在你的表单里面增加下面隐藏域：

```
<input type="hidden" name="__token__" value="{ $Request.token}" />
```

或者

```
{:token()}
```

然后在你的验证规则中，添加 `token` 验证规则即可，例如，如果使用的是验证器的话，可以改为：

```
protected $rule = [  
    'name' => 'require|max:25|token',  
    'email' => 'email',  
];
```

如果你的令牌名称不是 `__token__`，则表单需要改为：

```
<input type="hidden" name="__hash__" value="{ $Request.token.__hash__}" />
```

或者：

```
{:token('__hash')}
```

验证器中需要改为：

```
protected $rule = [  
    'name' => 'require|max:25|token:__hash__',  
    'email' => 'email',  
];
```

如果需要自定义令牌生成规则，可以调用 `Request` 类的 `token` 方法，例如：

```
namespace app\index\controller;  
  
use think\Controller;  
  
class Index extends Controller  
{  
    public function index()  
    {  
        $token = $this->request->token('__token__', 'sha1');  
        $this->assign('token', $token);  
    }  
}
```



```
        return $this->fetch();  
    }  
}
```

然后在模板表单中使用：

```
<input type="hidden" name="__token__" value="{${token}}" />
```

或者不需要在控制器写任何代码，直接在模板中使用：

```
{:token('__token__', 'sha1')}
```

杂项

[缓存](#)

[Session](#)

[Cookie](#)

[多语言](#)

[分页](#)

[上传](#)

[验证码](#)

[图像处理](#)

[文件处理](#)

[单元测试](#)

缓存

概述

ThinkPHP采用 `think\Cache` 类提供缓存功能支持。

设置

缓存支持采用驱动方式，所以缓存在使用之前，需要进行连接操作，也就是缓存初始化操作。

```
$options = [
    // 缓存类型为File
    'type' => 'File',
    // 缓存有效期为永久有效
    'expire'=> 0,
    //缓存前缀
    'prefix'=> 'think'
    // 指定缓存目录
    'path' => APP_PATH.'runtime/cache/',
];
Cache::connect($options);
```

或者通过定义配置参数的方式，在应用配置文件中添加：

```
'cache' => [
    'type' => 'File',
    'path' => CACHE_PATH,
    'prefix' => '',
    'expire' => 0,
],
```

支持的缓存类型包括file、memcache、wincache、sqlite、redis和xcache。

缓存参数根据不同的缓存方式会有所区别，通用的缓存参数如下：

| 参数 | 描述 |
|--------|--------------------|
| type | 缓存类型 |
| expire | 缓存有效期（默认为0 表示永久缓存） |
| prefix | 缓存前缀（默认为空） |

使用

缓存初始化之后，就可以进行相关缓存操作了。

如果通过配置文件方式定义缓存参数的话，可以无需手动进行缓存初始化操作，可以直接进行缓存读取和设置等操作。

设置缓存

缓存

设置缓存有效期

```
Cache::set('name',$value,3600);
```

如果设置成功返回true，否则返回false。

缓存自增

针对数值类型的缓存数据，可以使用自增操作，例如：

```
// name自增（步进值为1）
Cache::inc('name');
// name自增（步进值为3）
Cache::inc('name',3);
```

缓存自减

针对数值类型的缓存数据，可以使用自减操作，例如：

```
// name自减（步进值为1）
Cache::dec('name');
// name自减（步进值为3）
Cache::dec('name',3);
```

获取缓存

获取缓存数据可以使用：

```
dump(Cache::get('name'));
```

如果 `name` 值不存在，则默认返回 `false`。

支持指定默认值，例如：

```
dump(Cache::get('name',''));
```

表示如果 `name` 值不存在，则返回空字符串。

删除缓存

```
Cache::rm('name');
```

获取并删除缓存

```
Cache::pull('name');
```

如果 `name` 值不存在，则返回 `null`。

清空缓存

```
Cache::clear();
```

助手函数

系统对缓存操作提供了助手函数 `cache`，用法如下：

```
$options = [
    // 缓存类型为File
    'type' => 'File',
    // 缓存有效期为永久有效
    'expire' => 0,
    // 指定缓存目录
    'path' => APP_PATH . 'runtime/cache/',
];

// 缓存初始化
// 不进行缓存初始化的话，默认使用配置文件中的缓存配置
cache($options);

// 设置缓存数据
cache('name', $value, 3600);
// 获取缓存数据
var_dump(cache('name'));
// 删除缓存数据
cache('name', NULL);

// 设置缓存的同时并且进行参数设置
cache('test', $value, $options);
```

缓存标签

支持给缓存数据打标签，例如：

```
Cache::tag('tag')->set('name1', 'value1');
Cache::tag('tag')->set('name2', 'value2');
// 或者批量设置缓存标签
Cache::set('name1', 'value1');
Cache::set('name2', 'value2');
Cache::tag('tag', ['name1', 'name2']);
// 清除tag标签的缓存数据
Cache::clear('tag');
```

同时使用多个缓存类型

如果要同时使用多个缓存类型进行操作的话，可以做如下配置：

```
'cache' => [
    // 使用复合缓存类型
    'type' => 'complex',
    // 默认使用的缓存
    'default' => [
        // 驱动方式
        'type' => 'File',
        // 缓存保存目录
        'path' => CACHE_PATH,
```

```

],
// 文件缓存
'file' => [
  // 驱动方式
  'type' => 'file',
  // 设置不同的缓存保存目录
  'path' => RUNTIME_PATH . 'file/',
],
// redis缓存
'redis' => [
  // 驱动方式
  'type' => 'redis',
  // 服务器地址
  'host' => '127.0.0.1',
],
],
],

```

`cache.type` 配置为`complex`之后，就可以缓存多个缓存类型和缓存配置，每个缓存配置的方法和之前一样，并且你可以给相同类型的缓存类型（使用不同的缓存标识）配置不同的缓存配置参数。

当使用

```

Cache::set('name', 'value');
Cache::get('name');

```

的时候，其实使用的是 `default` 缓存标识的缓存配置，如果需要切换到其它的缓存标识操作，可以使用：

```

// 切换到file操作
Cache::store('file')->set('name', 'value');
Cache::get('name');
// 切换到redis操作
Cache::store('redis')->set('name', 'value');
Cache::get('name');

```

Session

概述

ThinkPHP采用 `think\Session` 类提供 `Session` 功能支持。

Session初始化

在ThinkPHP 5.0 中使用 `\think\Session` 类进行Session相关操作，Session会在第一次调用Session类的时候按照配置的参数自动初始化，例如，我们在应用配置中添加如下配置：

```
'session'                => [
    'prefix'              => 'think',
    'type'                 => '',
    'auto_start'          => true,
],
```

如果我们使用上述的session配置参数的话，无需任何操作就可以直接调用Session类的相关方法，例如：

```
Session::set('name','thinkphp');
Session::get('name');
```

如果你应用下面的不同模块需要不同的session参数，那么可以在模块配置文件中重新设置：

```
'session'                => [
    'prefix'              => 'module',
    'type'                 => '',
    'auto_start'          => true,
],
```

或者调用init方法进行初始化：

```
Session::init([
    'prefix'              => 'module',
    'type'                 => '',
    'auto_start'          => true,
]);
```

如果你没有使用Session类进行Session操作的话，例如直接操作 `$_SESSION`，必须使用上面的方式手动初始化或者直接调用 `session_start()` 方法进行 `session` 初始化。

设置参数

| 默认支持的session设置参数包括： | 参数 | 描述 |
|---------------------|-------------|----|
| type | session类型 | |
| expire | session过期时间 | |

| | |
|----------------|-----------------------|
| prefix | session前缀 |
| auto_start | 是否自动开启 |
| use_trans_sid | 是否使用use_trans_sid |
| var_session_id | 请求session_id变量名 |
| id | session_id |
| name | session_name |
| path | session保存路径 |
| domain | session cookie_domain |
| use_cookies | 是否使用cookie |
| cache_limiter | session_cache_limiter |
| cache_expire | session_cache_expire |

如果做了session驱动扩展，可能有些参数不一定有效。

基础用法

赋值

```
// 赋值（当前作用域）
Session::set('name','thinkphp');
// 赋值think作用域
Session::set('name','thinkphp','think');
```

判断是否存在

```
// 判断（当前作用域）是否赋值
Session::has('name');
// 判断think作用域下面是否赋值
Session::has('name','think');
```

取值

```
// 取值（当前作用域）
Session::get('name');
// 取值think作用域
Session::get('name','think');
```

如果name的值不存在，返回 `null`。

删除

```
// 删除（当前作用域）
Session::delete('name');
// 删除think作用域下面的值
Session::delete('name','think');
```


指定作用域

```
// 指定当前作用域
Session::prefix('think');
```

取值并删除

```
// 取值并删除
Session::pull('name');
```

如果name的值不存在，返回 **Null**。

清空

```
// 清除session（当前作用域）
Session::clear();
// 清除think作用域
Session::clear('think');
```

二级数组

支持session的二维数组操作，例如：

```
// 赋值（当前作用域）
Session::set('name.item', 'thinkphp');
// 判断（当前作用域）是否赋值
Session::has('name.item');
// 取值（当前作用域）
Session::get('name.item');
// 删除（当前作用域）
Session::delete('name.item');
```

助手函数

系统也提供了助手函数session完成相同的功能，例如：

```
// 初始化session
session([
    'prefix'    => 'module',
    'type'      => '',
    'auto_start' => true,
]);

// 赋值（当前作用域）
session('name', 'thinkphp');

// 赋值think作用域
session('name', 'thinkphp', 'think');

// 判断（当前作用域）是否赋值
session('?name');
```

```
// 取值（当前作用域）
session('name');

// 取值think作用域
session('name', '', 'think');

// 删除（当前作用域）
session('name', null);

// 清除session（当前作用域）
session(null);

// 清除think作用域
session(null, 'think');
```

Session驱动

支持指定 Session 驱动，配置文件如下：

```
'session' => [
  'prefix'      => 'module',
  'type'        => 'redis',
  'auto_start'  => true,
  // redis主机
  'host'        => '127.0.0.1',
  // redis端口
  'port'        => 6379,
  // 密码
  'password'    => '',
]
```

表示使用 `redis` 作为 `session` 类型。

Cookie

概述

ThinkPHP采用 `think\Cookie` 类提供Cookie支持。

基本操作

初始化

```
// cookie初始化
Cookie::init(['prefix'=>'think_', 'expire'=>3600, 'path'=>'/']);
// 指定当前前缀
Cookie::prefix('think_');
```

支持的参数及默认值如下：

```
// cookie 名称前缀
'prefix'    => '',
// cookie 保存时间
'expire'    => 0,
// cookie 保存路径
'path'      => '/',
// cookie 有效域名
'domain'    => '',
// cookie 启用安全传输
'secure'    => false,
// httponly设置
'httponly'  => '',
// 是否使用 setcookie
'setcookie' => true,
```

设置

```
// 设置Cookie 有效期为 3600秒
Cookie::set('name', 'value', 3600);
// 设置cookie 前缀为think_
Cookie::set('name', 'value', ['prefix'=>'think_', 'expire'=>3600]);
// 支持数组
Cookie::set('name', [1, 2, 3]);
```

判断

```
Cookie::has('name');
// 判断指定前缀的cookie值是否存在
Cookie::has('name', 'think_');
```

获取

```
Cookie::get('name');
// 获取指定前缀的cookie值
Cookie::get('name', 'think_');
```

删除

//删除cookie

```
Cookie::delete('name');  
// 删除指定前缀的cookie  
Cookie::delete('name', 'think_');
```

清空

```
// 清空指定前缀的cookie  
Cookie::clear('think_');
```

助手函数

系统提供了cookie助手函数用于基本的cookie操作，例如：

```
// 初始化  
cookie(['prefix' => 'think_', 'expire' => 3600]);  
  
// 设置  
cookie('name', 'value', 3600);  
  
// 获取  
echo cookie('name');  
  
// 删除  
cookie('name', null);  
  
// 清除  
cookie(null, 'think_');
```

多语言

ThinkPHP内置通过 `\think\Lang` 类提供多语言支持，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。

开启和加载语言包

要启用多语言功能，需要配置开启多语言行为，在应用的公共配置文件添加：

```
// 开启语言包功能
'lang_switch_on' => true,
// 支持的语言列表
'lang_list'      => ['zh-cn'],
```

开启多语言功能后，你可以在项目公共文件中设置要使用的语言，或者选择自动侦测当前语言。

```
// 设定当前语言
Lang::range('zh-cn');
// 或者进行自动检测语言
Lang::detect();
```

自动检测当前语言（主要是指浏览器访问的情况下）会对两种情况进行检测：

- 是否有 `$_GET['lang']`
- 识别 `$_SERVER['HTTP_ACCEPT_LANGUAGE']` 中的第一个语言
- 检测到任何一种情况下采用Cookie缓存
- 如果检测到的语言在`lang_list`配置参数之内认为有效，否则使用默认设置的语言

语言变量定义

语言变量的定义，只需要在需要使用多语言的地方，写成：

```
Lang::get('add user error');
// 使用系统封装的助手函数
lang('add user error');
```

也就是说，字符串信息要改成 `Lang::get` 方法来表示。

语言定义一般采用英语来描述。

语言文件定义

系统会默认加载下面两个语言包：

```
框架语言包： thinkphp\lang\当前语言.php
模块语言包： application\模块\lang\当前语言.php
```

如果你还需要加载其他的语言包，可以在设置或者自动检测语言之后，用load方法进行加载：

```
Lang::load(APP_PATH . 'common\lang\zh-cn.php');
```

ThinkPHP语言文件定义采用返回数组方式：

```
return [
    'hello thinkphp' => '欢迎使用ThinkPHP',
    'data type error' => '数据类型错误',
];
```

也可以在程序里面动态设置语言定义的值，使用下面的方式：

```
Lang::set('define2','语言定义');
$value = Lang::get('define2');
```

通常多语言的使用是在控制器里面，但是模型类的自动验证功能里面会用到提示信息，这个部分也可以使用多语言的特性。例如原来的方式是把提示信息直接写在模型里面定义：

```
['title','require','标题必须！',1],
```

如果使用了多语言功能的话（假设，我们在当前语言包里面定义了'lang_var'=>'标题必须！'），就可以这样定义模型的自动验证

```
['title','require','{%lang_var}',1],
```

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

```
{Think.lang.lang_var}
```

可以输出当前语言包里面定义的 `lang_var` 语言定义。

变量传入支持

语言包定义的时候支持传入变量，有两种方式

使用命名绑定方式，例如：

```
'file_format'    =>    '文件格式：{:format},文件大小：{:size}',
```

在模板中输出语言字符串的时候传入变量值即可：

```
{:lang('file_format',['format' => 'jpeg,png,gif,jpg','size' => '2MB'])}
```

第二种方式是使用格式字串，如果你需要使用第三方的翻译工具，建议使用该方式定义变量。

```
'file_format'    =>    '文件格式： %s,文件大小： %d',
```

在模板中输出多语言的方式更改为：

```
{:lang('file_format',['jpeg,png,gif,jpg','2MB'])}
```

分页

分页实现

ThinkPHP5.0 内置了分页实现，要给数据添加分页输出功能在 5.0 变得非常简单，可以直接在 Db 类查询的时候调用 `paginate` 方法：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = Db::name('user')->where('status',1)->paginate(10);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

也可以改成模型的分页查询代码：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

模板文件中分页输出代码如下：

```
<div>
<ul>
{volist name='list' id='user'}
    <li> {$user.nickname}</li>
{/volist}
</ul>
</div>
{$list->render()}
```

也可以单独赋值分页输出的模板变量

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10);
// 获取分页显示
$page = $list->render();
// 模板变量赋值
$this->assign('list', $list);
$this->assign('page', $page);
// 渲染模板输出
return $this->fetch();
```

模板文件中分页输出代码如下：

```
<div>
```



```
<ul>
{volist name='list' id='user'}
  <li> {$user.nickname}</li>
{/volist}
</ul>
</div>
{$page}
```

默认情况下，生成的分页输出是完整分页功能，带总分页数据和上下页码，分页样式只需要通过样式修改即可，完整分页默认生成的分页输出代码为：

```
<ul class="pagination">
<li><a href="?page=1">&laquo;</a></li>
<li><a href="?page=1">1</a></li>
<li class="active"><span>2</span></li>
<li class="disabled"><span>&raquo;</span></li>
</ul>
```

简洁分页

如果你仅仅需要输出一个 仅仅只有上下页的分页输出，可以使用下面的简洁分页代码：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10,true);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

简洁分页模式的输出代码为：

```
<ul class="pager">
<li><a href="?page=1">&laquo;</a></li>
<li class="disabled"><span>&raquo;</span></li>
</ul>
```

由于简洁分页模式不需要查询总数据数，因此可以提高查询性能。

分页参数

| 主要的分页参数如下： | 参数 | 描述 |
|------------|---------|----|
| list_rows | 每页数量 | |
| page | 当前页 | |
| path | url路径 | |
| query | url额外参数 | |
| fragment | url锚点 | |
| var_page | 分页变量 | |
| type | 分页类名 | |

分页参数的设置方式有两种，第一种是在配置文件中定义，例如：

```
//分页配置
'paginate' => [
  'type' => 'bootstrap',
  'var_page' => 'page',
],
```

type属性支持命名空间，例如：

```
//分页配置
'paginate' => [
  'type' => '\org\page\bootstrap',
  'var_page' => 'page',
],
```

也可以在调用分页方法的时候传入，例如：

```
$list = Db::name('user')->where('status',1)->paginate(10,true,[
  'type' => 'bootstrap',
  'var_page' => 'page',
]);
```

上传

上传文件

ThinkPHP5.0 对文件上传的支持更加简单。

内置的上传只是上传到本地服务器，上传到远程或者第三方平台的话需要自己扩展。

假设表单代码如下：

```
<form action="/index/index/upload" enctype="multipart/form-data" method="post">
<input type="file" name="image" /> <br>
<input type="submit" value="上传" />
</form>
```

然后在控制器中添加如下的代码：

```
public function upload(){
    // 获取表单上传文件 例如上传了001.jpg
    $file = request()->file('image');
    // 移动到框架应用根目录/public/uploads/ 目录下
    $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
    if($info){
        // 成功上传后 获取上传信息
        // 输出 jpg
        echo $info->getExtension();
        // 输出 42a79759f284b767dfcb2a0197904287.jpg
        echo $info->getFilename();
    }else{
        // 上传失败获取错误信息
        echo $file->getError();
    }
}
```

move 方法成功的话返回的是一个 SplFileInfo 对象，你可以对上传后的文件进行后续操作。

多文件上传

如果你使用的是多文件上传表单，例如：

```
<form action="/index/index/upload" enctype="multipart/form-data" method="post">
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<input type="submit" value="上传" />
</form>
```

控制器代码可以改成：

```
public function upload(){
    // 获取表单上传文件
    $files = request()->file('image');
    foreach($files as $file){
        // 移动到框架应用根目录/public/uploads/ 目录下
        $info = $file->move(ROOT_PATH . 'public' . DS . 'uploads');
        if($info){
            // 成功上传后 获取上传信息
            // 输出 jpg
            echo $info->getExtension();
            // 输出 42a79759f284b767dfcb2a0197904287.jpg
            echo $info->getFilename();
        }else{
            // 上传失败获取错误信息
            echo $file->getError();
        }
    }
}
```

上传规则

默认情况下，会在上传目录下面生成以当前日期为子目录，以微秒时间的 md5 编码为文件名的文件，例如上面生成的文件名可能是：

```
/home/www/upload/20160510/42a79759f284b767dfcb2a0197904287.jpg
```

我们可以指定上传文件的命名规则，使用 rule 方法即可，例如：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用md5规则
$file->rule('md5')->move('/home/www/upload/');
```

最终生成的文件名类似于：

```
/home/www/upload/72/ef580909368d824e899f77c7c98388.jpg
```

| 系统默认提供了几种上传命名规则，包括： | 规则 | 描述 |
|---------------------|--------------------|----|
| date | 根据日期和微秒数生成 | |
| md5 | 对文件使用md5_file散列生成 | |
| sha1 | 对文件使用sha1_file散列生成 | |

其中md5和sha1规则会自动以散列值的前两个字符作为子目录，后面的散列值作为文件名。

如果需要使用自定义命名规则，可以在 rule 方法中传入函数或者方法，例如：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
```

```
// 移动到服务器的上传目录 并且使用uniqid规则
$file->rule('uniqid')->move('/home/www/upload/');
```

生成的文件名类似于：

```
/home/www/upload/573d3b6d7abe2.jpg
```

如果你希望保留原文件名称，可以使用：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用原文件名
$file->move('/home/www/upload/', '');
```

默认情况下，会覆盖服务器上传目录下的同名文件，如果不希望覆盖，可以使用：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且设置不覆盖
$file->move('/home/www/upload/', true, false);
```

返回对象

上传成功后返回的仍然是一个 **File** 对象，除了File对象自身的方法外，并且可以使用 **SplFileObject** 的属性和方法，便于进行后续的文件处理。

验证码

首先使用 Composer 安装 think-captcha 扩展包：

```
composer require tophink/think-captcha
```

验证码配置

然后在应用配置文件中添加验证码的配置参数

```
'captcha' => [
    // 验证码字符集合
    'codeSet' => '2345678abcdefghijkmpqrstuvwxyABCDEFGHIJKLMNOPQRSTUVWXYZ',
    // 验证码字体大小(px)
    'fontSize' => 25,
    // 是否画混淆曲线
    'useCurve' => true,
    // 验证码图片高度
    'imageH' => 30,
    // 验证码图片宽度
    'imageW' => 100,
    // 验证码位数
    'length' => 5,
    // 验证成功后是否重置
    'reset' => true
],
```

并且确保开启了URL路由。

验证码显示

```
<div>{:captcha_img()}</div>
```

或者

```
<div></div>
```

上面两种的最终效果是一样的，根据需要调用即可。

控制器验证

使用TP5的内置验证功能，添加 captcha 验证规则即可

```
$this->validate($data, [
    'captcha|验证码'=>'require|captcha'
]);
```

验证码

或者手动验证

```
if(!captcha_check($captcha)){  
    //验证失败  
};
```

图像处理

安装扩展

使用 Composer 安装 ThinkPHP5 的图像处理类库：

```
composer require topthink/think-image
```

图像操作

下面来看下图像操作类的基础方法。

打开图像文件

假设当前入口文件目录下面有一个 `image.png` 文件，如图所示：



使用 `open` 方法打开图像文件进行相关操作：

```
$image = \think\Image::open('./image.png');
```

也可以从直接获取当前请求中的文件上传对象，例如：

```
$image = \think\Image::open(request()->file('image'));
```

获取图像信息

本文档使用 [看云](#) 构建

可以获取打开图片的信息，包括图像大小、类型等，例如：

```
$image = \think\Image::open('./image.png');  
// 返回图片的宽度  
$width = $image->width();  
// 返回图片的高度  
$height = $image->height();  
// 返回图片的类型  
$type = $image->type();  
// 返回图片的mime类型  
$mime = $image->mime();  
// 返回图片的尺寸数组 0 图片宽度 1 图片高度  
$size = $image->size();
```

裁剪图片

使用 `crop` 和 `save` 方法完成裁剪图片功能。

```
$image = \think\Image::open('./image.png');  
//将图片裁剪为300x300并保存为crop.png  
$image->crop(300, 300)->save('./crop.png');
```

生成的图片如图：



支持从某个坐标开始裁剪，例如下面从（100，30）开始裁剪，例如：

```
$image = \think\Image::open('./image.png');  
//将图片裁剪为300x300并保存为crop.png  
$image->crop(300, 300, 100, 30)->save('./crop.png');
```

生成的图片如图：



生成缩略图

使用 `thumb` 方法生成缩略图，例如：

```
$image = \think\Image::open('./image.png');  
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png  
$image->thumb(150, 150)->save('./thumb.png');
```

生成的缩略图如图所示：

我们看到实际生成的缩略图并不是150*150，因为默认采用原图等比例缩放的方式生成缩略图，最大宽度是150。

可以支持其他类型的缩略图生成，设置包括 `\think\Image` 的下列常量或者对应的数字：

```
//常量，标识缩略图等比例缩放类型  
const THUMB_SCALING    = 1;  
//常量，标识缩略图缩放后填充类型  
const THUMB_FILLED     = 2;  
//常量，标识缩略图居中裁剪类型  
const THUMB_CENTER     = 3;  
//常量，标识缩略图左上角裁剪类型  
const THUMB_NORTHWEST = 4;  
//常量，标识缩略图右下角裁剪类型  
const THUMB_SOUTHEAST = 5;  
//常量，标识缩略图固定尺寸缩放类型  
const THUMB_FIXED      = 6;
```

比如我们居中裁剪：

```
$image = \think\Image::open('./image.png');  
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png  
$image->thumb(150, 150, \think\Image::THUMB_CENTER)->save('./thumb.png');
```

后生成的缩略图效果如图：



再比如我们右下角剪裁

```
$image = \think\Image::open('./image.png');  
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png  
$image->thumb(150,150,\think\Image::THUMB_SOUTHEAST)->save('./thumb.png');
```

生成的缩略图效果如图：



这里就不再对其他用法一一举例了。

图像翻转

使用 `flip` 可以对图像进行翻转操作，默认是以x轴进行翻转，例如：

```
$image = \think\Image::open('./image.png');  
// 对图像进行以x轴进行翻转操作  
$image->flip()->save('./flip_image.png');
```

生成的效果如图：



我们也可以改变参数，以y轴进行翻转，例如：

```
$image = \think\Image::open('./image.png');  
// 对图像进行以y轴进行翻转操作  
$image->flip(\think\image::FLIP_Y)->save('./flip_image.png');
```

生成的效果如图：



图像的翻转可以理解为图像的镜面效果与图像旋转有所不同。

图像旋转

使用 `rotate` 可以对图像进行旋转操作（默认是顺时针旋转90度），我们用默认90度进行旋转举例：

```
$image = \think\Image::open('./image.png');  
// 对图像使用默认的顺时针旋转90度操作  
$image->rotate()->save('./rotate_image.png');
```

生成的效果如图：



图像保存参数

| save 方法可以配置的参数 | 参数 | 默认 | 描述 |
|----------------|---------|-------------------|----|
| pathname | 必填项 | 图像保存路径名称 | |
| type | 默认与原图相同 | 图像类型 | |
| quality | 80 | 图像质量 | |
| interlace | true | 是否对JPEG类型图像设置隔行扫描 | |

设置隔行扫描的情况下在网页进行浏览时。是从上到下一行一行的显示，否则图片整个显示出来 然后由模糊到清晰显示。

添加水印

系统支持添加图片及文字水印，下面依次举例说明

添加图片水印，我们下载官网logo文件到根目录进行举例：

```
$image = \think\Image::open('./image.png');  
// 给原图左上角添加水印并保存water_image.png  
$image->water('./logo.png')->save('water_image.png');
```

`water` 方法的第二个参数表示水印的位置，默认值是 `WATER_SOUTH`，可以传入下列 `\think\Image` 类的常量或者对应的数字：

```
//常量，标识左上角水印  
const WATER_NORTHWEST = 1;  
//常量，标识上居中水印  
const WATER_NORTH = 2;  
//常量，标识右上角水印  
const WATER_NORTHEAST = 3;  
//常量，标识左居中水印  
const WATER_WEST = 4;  
//常量，标识居中水印  
const WATER_CENTER = 5;  
//常量，标识右居中水印  
const WATER_EAST = 6;  
//常量，标识左下角水印  
const WATER_SOUTHWEST = 7;  
//常量，标识下居中水印  
const WATER_SOUTH = 8;  
//常量，标识右下角水印  
const WATER_SOUTHEAST = 9;
```

我们用左上角来进行测试：

```
$image = \think\Image::open('./image.png');  
// 给原图左上角添加水印并保存water_image.png  
$image->water('./logo.png',\think\Image::WATER_NORTHWEST)->save('water_image.png');
```

生成的图片效果如下：



还可以支持水印图片的透明度（0~100，默认值是100），例如：

```
$image = \think\Image::open('./image.png');  
// 给原图左上角添加透明度为50的水印并保存alpha_image.png  
$image->water('./logo.png', \think\Image::WATER_NORTHWEST, 50)->save('alpha_image.png');
```

生成的图片效果如下：



也可以支持给图片添加文字水印（我们复制一个字体文件 HYQingKongTiJ.ttf 到入口目录），我们现在生成一个像素20px，颜色为 #ffffff 的水印效果：

```
$image = \think\Image::open('./image.png');  
// 给原图左上角添加水印并保存water_image.png  
$image->text('十年磨一剑 - 为API开发设计的高性能框架', 'HYQingKongTiJ.ttf', 20, '#ffffff')->save('text_image.png');
```

生成的图片效果：



文字水印参数

文字水印比较多，在此只做说明不做演示了

| 参数 | 默认 | 描述 |
|--------|-----------------|--------------|
| text | 不能为空 | 添加的文字 |
| font | 不能为空 | 字体文件路径 |
| size | 不能为空 | 字号，单位是像素 |
| color | #00000000 | 文字颜色 |
| locate | WATER_SOUTHEAST | 文字写入位置 |
| offset | 0 | 文字相对当前位置的偏移量 |
| angle | 0 | 文字倾斜角度 |

文件处理

ThinkPHP5.0 内置了一个文件处理类 `\think\File`，内置的文件上传操作也是调用了该类进行处理的。

`File` 类继承了PHP的 `SplFileObject` 类，因此可以调用 `SplFileObject` 类所有的属性和方法。

单元测试

单元测试

首先安装 ThinkPHP5 的单元测试扩展，进入命令行，切换到tp5的应用根目录下面，执行：

```
composer require topthink/think-testing
```

由于单元测试扩展的依赖较多，因此安装过程会比较久，请耐心等待。

安装完成后，会在应用根目录下面增加 `tests` 目录和 `phpunit.xml` 文件。

默认带了一个 `tests/ExampleTest.php` 单元测试文件，我们可以直接在命令行下面运行单元测试：

```
php think unit
```

请始终使用以上命令进行单元测试，而不是直接用 `phpunit` 来运行单元测试。

添加单元测试文件

我们来添加一个新的单元测试文件，单元测试文件为 `tests/IndexTest.php`，内容如下：

```
<?php
use tests\TestCase;

class IndexTest extends TestCase
{
    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }
}
```

注意，单元测试文件中定义的测试类如果不存在冲突，可以不需要使用命名空间。

扩展

[函数](#)

[类库](#)

[行为](#)

[驱动](#)

[Composer包](#)

[SAE](#)

[标签扩展](#)

函数

你可以方便的在 **ThinkPHP5** 中添加自定义函数，包括替换已有的助手函数。

扩展系统函数

这里指的系统函数是对系统自带的函数进行替换或者增加，具体方式如下：

在应用目录下面增加一个助手函数文件（文件名随意，例如 **application/helper.php** ），添加：

```
// 增加一个新的table助手函数
function table($table, $config = [])
{
    return \think\Db::connect($config)->setTable($table);
}

// 替换已有的db助手函数
function db($name, $config= [])
{
    return \think\Db::connect($config)->name($name);
}
```

然后，在应用配置文件中设置：

```
// 扩展函数文件
'extra_file_list'          => [ APP_PATH . 'helper.php', THINK_PATH . 'helper.php'],
```

extra_file_list 定义的顺序不能反，否则就不能替换已有的助手函数。

注意，尽量避免直接修改核心的 **helper.php** 文件。

添加应用函数

如果需要给当前应用添加函数，只需要在应用的公共文件（ **application/common.php** ）中定义需要的函数即可，系统会自动加载，如果你需要增加新的函数文件，例如需要增加一个 **sys.php**，那么就需要和上面一样设置 **extra_file_list** 配置：

```
// 扩展函数文件
'extra_file_list'          => [ APP_PATH . 'helper.php', THINK_PATH . 'helper.php', APP_P
ATH . 'sys.php'],
```

类库

如果你需要在核心之外扩展和使用第三方类库，并且该类库不是通过 **Composer** 安装使用，那么可以直接放入应用根目录下面的 **extend** 目录下面，该目录是官方建议的第三方扩展类库目录。

类的命名规范遵循 **PSR-2** 及 **PSR-4** 规范，例如，如果有一个扩展类库的命名空间是 **first.second.Foo**，类定义如下：

```
namespace first\second;

class Foo
{
}
```

使用第三方类库的时候注意不要和系统的命名空间产生冲突，例如核心的 **think**、**app** 以及 **Composer** 类库自身定义的命名空间。

那么实际的类文件位置应该是：

```
extend/first/second/Foo.php
```

使用 **first.second.Foo** 类的时候，直接实例化即可使用，例如：

```
$foo = new \first\second\Foo();
```

或者先

```
use first\second\Foo;
```

然后

```
$foo = new Foo();
```

你可以在入口文件中随意修改 **extend** 目录的名称，例如：

```
define('EXTEND_PATH', '../extension/');
```

ThinkPHP5 建议所有的扩展类库都使用命名空间定义，如果你的类库没有使用命名空间，则不支持自动加载，必须使用 **Loader::import** 方法先导入文件后才能使用。

```
Loader::import('first.second.Foo');  
$foo = new \Foo();
```

强烈建议使用 Composer 安装和更新扩展类库，ThinkPHP5.0 的扩展类库都采用 Composer 方式进行安装。

行为

概述

行为（Behavior）是ThinkPHP扩展机制中比较关键的一项扩展，行为既可以独立调用，也可以绑定到某个标签中进行侦听，在官方提出的CBD模式中行为也占了主要的地位，可见行为在ThinkPHP框架中意义非凡。

这里指的行为是一个比较抽象的概念，你可以把行为想象成在应用执行过程中的一个动作或者处理。在框架的执行流程中，例如路由检测是一个行为，静态缓存是一个行为，用户权限检测也是行为，大到业务逻辑，小到浏览器检测、多语言检测等等都可以当做是一个行为，甚至说你希望给你的网站用户的第一次访问弹出Hello，world！这些都可以看成是一种行为，行为的存在让你无需改动框架和应用，而在外围通过扩展或者配置来改变或者增加一些功能。

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

行为标签位

| 在定义行为之前，我们先来了解下系统有哪些标签位，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 |
| app_begin | 应用开始标签位 |
| module_init | 模块初始化标签位 |
| action_begin | 控制器开始标签位 |
| view_filter | 视图输出过滤标签位 |
| app_end | 应用结束标签位 |
| log_write | 日志write方法标签位 |

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。

添加行为标签位

可以使用 `\think\Hook` 类的 `listen` 方法添加自己的行为侦听位置，例如：

```
Hook::listen('action_init');
```

可以给侦听方法传入参数（仅能传入一个参数），该参数使用引用传值，因此必须使用变量，例如：

```
Hook::listen('action_init',$params);
```

侦听的标签位置可以随意放置。

行为定义

行为类的定义很简单，定义行为的执行入口方法 `run` 即可，例如：

```
namespace app\index\behavior;

class Test
{
    public function run(&$params)
    {
        // 行为逻辑
    }
}
```

行为类并不需要继承任何类，相对比较灵活。

如果行为类需要绑定到多个标签，可以采用如下定义：

```
namespace app\index\behavior;

class Test
{
    public function app_init(&$params)
    {

    }

    public function app_end(&$params)
    {

    }
}
```

绑定到 `app_init` 和 `app_end` 后 就会调用相关的方法。

行为绑定

行为定义完成后，就需要绑定到某个标签位置才能生效，否则是不会执行的。

使用Hook类的add方法注册行为，例如：

```
// 注册 app\\index\\behavior\\CheckLang行为类到app_init标签位
```

```
Hook::add('app_init','app\\index\\behavior\\CheckLang');
//注册 app\\admin\\behavior\\CronRun行为类到app_init标签位
Hook::add('app_init','app\\admin\\behavior\\CronRun');
```

如果要批量注册行为的话，可以使用：

```
Hook::add('app_init',['app\\index\\behavior\\CheckAuth','app\\index\\behavior\\CheckLang','app\\admin\\behavior\\CronRun']);
```

当应用运行到 `app_init` 标签位的时候，就会依次调用 `app\\index\\behavior\\CheckAuth`、`app\\index\\behavior\\CheckLang` 和 `app\\admin\\behavior\\CronRun` 行为。如果其中一个行为中有中止代码的话则后续不会执行，如果返回 `false` 则当前标签位的后续行为将不会执行，但应用将继续运行。

我们也可以直接在 `APP_PATH` 目录下面或者模块的目录下面定义 `tags.php` 文件来统一定义行为，定义格式如下：

```
return [
    'app_init'=> [
        'app\\index\\behavior\\CheckAuth',
        'app\\index\\behavior\\CheckLang'
    ],
    'app_end'=> [
        'app\\admin\\behavior\\CronRun'
    ]
]
```

如果 `APP_PATH` 目录下面和模块目录下面的 `tags.php` 都定义了 `app_init` 的行为绑定的话，会采用合并模式，如果希望覆盖，那么可以在模块目录下面的 `tags.php` 中定义如下：

```
return [
    'app_init'=> [
        'app\\index\\behavior\\CheckAuth',
        '_overlay'=>true
    ],
    'app_end'=> [
        'app\\admin\\behavior\\CronRun'
    ]
]
```

如果某个行为标签定义了 `'_overlay' =>true` 就表示覆盖之前的相同标签下面的行为定义。

闭包支持

可以不用定义行为直接把闭包函数绑定到某个标签位，例如：

```
Hook::add('app_init',function(){
    echo 'Hello,world!';
});
```

如果标签位有传入参数的话，闭包也可以支持传入参数，例如：

```
Hook::listen('action_init',$params);
Hook::add('action_init',function($params){
    var_dump($params);
});
```

直接执行行为

如果需要，你也可以不绑定行为标签，直接调用某个行为，使用：

```
// 执行 app\index\behavior\CheckAuth行为类的run方法 并引用传入params参数
$result = Hook::exec('app\\index\\behavior\\CheckAuth','run',$params);
```

驱动

系统的驱动类都支持单独扩展，并且驱动文件的位置和命名空间可以随意设置，包括缓存、日志、调试和数据库驱动。

以缓存驱动为例，如果我们扩展了一个自己的 `redis` 驱动，类名为 `app\driver\cache\Redis`，那么我们只需要设置缓存类型为：

```
'cache' => [  
  // 驱动方式  
  'type' => '\app\driver\cache\Redis',  
  // 缓存前缀  
  'prefix' => '',  
  // 缓存有效期 0表示永久缓存  
  'expire' => 0,  
],
```

Composer包

新版建议采用 **Composer** 包的方式扩展框架及类库，关于 **Composer** 的基础知识请参考 [Composer 官方文档（英文）](#)，或者在看云上阅读本 [中文版本](#)。

Composer安装

ThinkPHP5.0支持使用Composer安装包，例如在应用根目录下面执行：

```
composer require topthink/think-mongo
```

更新

```
composer update topthink/think-mongo
```

Time

时间戳操作

首先通过 composer 安装

```
composer require topthink/think-helper
```

在文件头部引入

```
use think\helper\Time;
```

比如需要获得今天的零点时间戳和23点59分59秒的时间戳

```
list($start, $end) = Time::today();  
  
echo $start; // 零点时间戳  
echo $end; // 23点59分59秒的时间戳
```

完整示例如下:

```
// 今日开始和结束的时间戳  
Time::today();  
  
// 昨日开始和结束的时间戳  
Time::yesterday();  
  
// 本周开始和结束的时间戳  
Time::week();  
  
// 上周开始和结束的时间戳  
Time::lastWeek();  
  
// 本月开始和结束的时间戳  
Time::month();  
  
// 上月开始和结束的时间戳  
Time::lastMonth();  
  
// 今年开始和结束的时间戳  
Time::year();  
  
// 去年开始和结束的时间戳  
Time::lastYear();  
  
// 获取7天前零点到现在的时间戳  
Time::dayToNow(7)  
  
// 获取7天前零点到昨日结束的时间戳  
Time::dayToNow(7, true)  
  
// 获取7天前的时间戳
```

```
Time::daysAgo(7)

// 获取7天后的时间戳
Time::daysAfter(7)

// 天数转换成秒数
Time::daysToSecond(5)

// 周数转换成秒数
Time::weekToSecond(5)
```


SAE

SAE介绍

Sina App Engine (简称SAE) 是新浪研发中心开发的国内首个公有云计算平台，是新浪云计算战略的核心组成部分，作为一个简单高效的分布式Web服务开发、运行平台越来越受开发者青睐。

SAE环境和普通环境有所不同，它是一个分布式服务器集群，能让你的程序同时运行在多台服务器中。并提供了很多高效的分布式服务。SAE为了提升性能和安全，禁止了本地IO写操作，使用MemcacheX、Storage等存储型服务代替传统IO操作，效率比传统IO读写操作高，有效解决因IO瓶颈导致程序性能低下的问题。

正是因为SAE和普通环境的不同，使得普通程序不能直接放在SAE上，需要经过移植才能放在SAE上运行。也使得很多能在SAE上运行的程序不能在普通环境下运行。

thinkphp5.0支持了SAE的扩展，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。

安装SAE扩展

```
composer require tophink/think-sae
```

相关配置

数据库配置

数据库配置文件 `database.php` 中修改为：

```
// 数据库类型
'type' => 'mysql',
// 服务器地址
'hostname' => SAE_MYSQL_HOST_M . ',' . SAE_MYSQL_HOST_S,
// 数据库名
'database' => SAE_MYSQL_DB,
// 用户名
'username' => SAE_MYSQL_USER,
// 密码
'password' => SAE_MYSQL_PASS,
// 端口
'hostport' => SAE_MYSQL_PORT,
```

驱动配置

```
'log' => [
    'type' => '\think\sae\Log',
]

'template' => [
    'type' => 'Think',
    'compile_type' => '\think\sae\Template',
```

```
]
'cache'    => [
  'type'    => '\\think\sae\Cache',
]
```

标签扩展

标签库加载

模板中加载标签库，预加载自定义标签库，扩展内置标签库的加载 请参考:模板/标签库

建议开发者将自定义标签库请放置应用目录中，请勿放在框架系统目录内，以免使用 [Composer](#) 更新框架时导致自定义标签库的丢失

下面以标签库放在 `common` 作为一个示例：

```
<?php
namespace app\common>taglib;
use think\template\TagLib;
class Demo extends TagLib{
    /**
     * 定义标签列表
     */
    protected $tags = [
        // 标签定义：attr 属性列表 close 是否闭合（0 或者1 默认1） alias 标签别名 level 嵌套层
        'close'      => ['attr' => 'time,format', 'close' => 0], //闭合标签，默认为不闭合
        'open'       => ['attr' => 'name,type', 'close' => 1],

    ];

    /**
     * 这是一个闭合标签的简单演示
     */
    public function tagClose($tag)
    {
        $format = empty($tag['format']) ? 'Y-m-d H:i:s' : $tag['format'];
        $time = empty($tag['time']) ? time() : $tag['time'];
        $parse = '<?php ';
        $parse .= 'echo date("' . $format . '",' . $time . ');';
        $parse .= ' ?>';
        return $parse;
    }

    /**
     * 这是一个非闭合标签的简单演示
     */
    public function tagOpen($tag, $content)
    {
        $type = empty($tag['type']) ? 0 : 1; // 这个type目的是为了区分类型，一般来源是数据库
        $name = $tag['name']; // name是必填项，这里不做判断了
        $parse = '<?php ';
        $parse .= '$test_arr=[[1,3,5,7,9],[2,4,6,8,10]]'; // 这里是模拟数据
        $parse .= '$__LIST__ = $test_arr[' . $type . ']';
        $parse .= ' ?>';
        $parse .= '{volist name="__LIST__" id="' . $name . '"}';
        $parse .= $content;
        $parse .= '{/volist}';
        return $parse;
    }
}
```

```
}
```

这时候我们的控制器继承 `Controller` ,在配置参数中配置：

```
'template' => [
    // 模板引擎类型 支持 php think 支持扩展
    'type' => 'Think',
    // 模板路径
    'view_path' => '',
    // 模板后缀
    'view_suffix' => '.html',
    // 预先加载的标签库
    'taglib_pre_load' => 'app\common\taglib\Demo',
],
```

我们就可以在控制器中对模版赋值：

```
//给模版给以一个当前时间戳的值
$this->assign('demo_time',$this->request->time());
```

在模版中调用我们已经预先加载的标签：

```
<h1>闭合标签</h1>
{demo:close time='$demo_time' /}
<hr>
<h1>非闭合标签</h1>
{demo:open name='demo_name'}
    {$key}=>{$demo_name}<br>
{/demo:open}
<br>
{demo:open name='demo_name' type='1'}
    {$key}=>{$demo_name}<br>
{/demo:open}
```

关于标签库开发

暂时可以参考3.2的官方手册中关于标签库扩展的部分，建议自己分析内置标签库Cx。

命令行

ThinkPHP5.0支持 **Console** 应用，通过命令行的方式执行一些URL访问不方便或者安全性较高的操作。

我们可以在命令行下面，切换到应用根目录，然后执行 `php think`，会出现下面的提示信息：

```
>php think
Think Console version 0.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -V, --version          Display this console version
  -q, --quiet            Do not output any message
  --ansi                Force ANSI output
  --no-ansi              Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal output, 2 for
more verbose output and 3 for debug

Available commands:
  build                Build Application Dirs
  clear                Clear runtime file
  help                Displays help for a command
  list                Lists commands
  make
  make:controller      Create a new resource controller class
  make:model           Create a new model class
  optimize
  optimize:autoload    Optimizes PSR0 and PSR4 packages to be loaded with classmaps too,
good for production.
  optimize:config      Build config and common file cache.
  optimize:route       Build route cache.
```

`console` 命令的执行格式一般为：

>php think 指令 参数

下面介绍下系统自带的几个命令，包括：

| 指令 | 描述 |
|-------------------|-----------|
| build | 自动生成目录和文件 |
| help | 帮助 |
| list | 指令列表 |
| clear | 清除缓存指令 |
| make:controller | 创建控制器文件 |
| make:model | 创建模型文件 |
| optimize:autoload | 生成类库映射文件 |

命令行

| | |
|-----------------|----------|
| optimize:config | 生成配置缓存文件 |
| optimize:route | 生成路由缓存文件 |

更多的指令可以自己扩展。

自动生成目录结构

ThinkPHP5.0 具备自动创建功能，可以用来自动生成需要的模块及目录结构和文件等，自动生成主要调用 `\think\Build` 类库。

生成规则定义

首先需要定义一个用于自动生成的规则定义文件，通常命名为 `build.php`。

默认的框架的根目录下面自带了一个 `build.php` 示例参考文件，内容如下：

```
return [
    // 生成运行时目录
    '__file__' => ['common.php'],

    // 定义index模块的自动生成
    'index' => [
        '__file__' => ['common.php'],
        '__dir__' => ['behavior', 'controller', 'model', 'view'],
        'controller' => ['Index', 'Test', 'UserType'],
        'model' => [],
        'view' => ['index/index'],
    ],
    // ... 其他更多的模块定义
];
```

可以给每个模块定义需要自动生成的文件和目录，以及MVC类。

- `__dir__` 表示生成目录（支持多级目录）
- `__file__` 表示生成文件（不定义默认会生成 `config.php` 文件）
- `controller` 表示生成controller类
- `model`表示生成model类
- `view`表示生成html文件（支持子目录）

自动生成以 `APP_PATH` 为起始目录，`__dir__` 和 `__file__` 表示需要自动创建目录和文件，其他的则表示为模块自动生成。

模块的自动生成则以 `APP_PATH. '模块名/'` 为起始目录。

并且会自动生成模块的默认的Index访问控制器文件用于显示框架的欢迎页面。

我们还可以在 `APP_PATH` 目录下面自动生成其它的文件和目录，或者增加多个模块的自动生成，例如：

```
return [
    '__file__' => ['hello.php', 'test.php'],
    // 定义index模块的自动生成
    'index' => [
        '__file__' => ['tags.php', 'user.php', 'hello.php'],
        '__dir__' => ['behavior', 'controller', 'model', 'view'],
    ],
    // ... 其他更多的模块定义
];
```

```

        'controller' => ['Index', 'Test', 'UserType'],
        'model'      => [],
        'view'       => ['index/index'],
    ],
    // 定义test模块的自动生成
    'test'=>[
        '__dir__'    => ['behavior', 'controller', 'model', 'widget'],
        'controller'=> ['Index', 'Test', 'UserType'],
        'model'      => ['User', 'UserType'],
        'view'       => ['index/index', 'index/test'],
    ],
];

```

命令行自动生成

我们通过控制台来完成自动生成，切换到命令行，在应用的根目录输入下面命令：

```
>php think build
```

如果看到输出

```
Successed
```

则表示自动生成成功。

默认会读取应用目录 `application` 下面的 `build.php` 作为自动生成的定义文件，如果你的定义文件位置不同，则需要使用 `--config` 参数指定如下：

```
>php think build --config build.php
```

表示读取根目录下的 `build.php` 文件。

生成模块指令

```
>php think build --module test
```

表示自动生成 `test` 模块。

添加自动生成代码

如果你不习惯命令行操作，也可以直接调用 `\think\Build` 类的方法进行自动生成，例如：

```

// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
// 读取自动生成定义文件
$build = include 'build.php';
// 运行自动生成
\think\Build::run($build);

```


`run` 方法第二个参数用于指定要生成的应用类库的命名空间，默认是 `app`，第三个参数是设置是否需要使用类后缀。

例如：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
// 读取自动生成定义文件
$build = include 'build.php';
// 运行自动生成
\think\Build::run($build, 'application', true);
```

可以不依赖自动生成文件，直接使用默认目录生成模块，例如：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
// 自动生成admin模块
\think\Build::module('admin');
```

`module` 方法第二个参数和第三个参数的用法和 `run` 方法一样。

创建类库文件

快速生成控制器类

执行下面的指令可以生成 `index` 模块的 `Blog` 控制器类库文件

```
>php think make:controller index/Blog
```

生成的控制器类文件如下：

```
<?php

namespace app\index\controller;

use think\Controller;
use think\Request;

class Blog extends Controller
{
    /**
     * 显示资源列表
     *
     * @return \think\Response
     */
    public function index()
    {
        //
    }

    /**
     * 显示创建资源表单页。
     *
     * @return \think\Response
     */
    public function create()
    {
        //
    }

    /**
     * 保存新建的资源
     *
     * @param \think\Request $request
     * @return \think\Response
     */
    public function save(Request $request)
    {
        //
    }

    /**
     * 显示指定的资源
     *
     * @param int $id
     * @return \think\Response
     */
}
```

```

public function read($id)
{
    //
}

/**
 * 显示编辑资源表单页.
 *
 * @param int $id
 * @return \think\Response
 */
public function edit($id)
{
    //
}

/**
 * 保存更新的资源
 *
 * @param \think\Request $request
 * @param int $id
 * @return \think\Response
 */
public function update(Request $request, $id)
{
    //
}

/**
 * 删除指定资源
 *
 * @param int $id
 * @return \think\Response
 */
public function delete($id)
{
    //
}
}

```

默认生成的控制器类继承 `\think\Controller`，并且生成了资源操作方法，如果仅仅生成空的控制器则可以使用：

```
>php think make:controller index\Blog --plain
```

快速生成模型类

执行下面的指令可以生成 `index` 模块的 `Blog` 模型类库文件

```
>php think make:model index/Blog
```

生成的模型类文件如下：

```

namespace app\index\model;

use think\Model;

```

```
class Blog extends Model
{
}
```

生成类库映射文件

可以使用下面的指令生成类库映射文件，提高系统自动加载的性能。

```
>php think optimize:autoload
```

指令执行成功后，会在runtime目录下面生成 `classmap.php` 文件，生成的类库映射文件会扫描系统目录和应用目录的类库。

生成路由缓存

如果你的应用定义了比较多的路由规则，可以使用下面的指令生成路由缓存文件，提高系统的路由检测的性能。

```
>php think optimize:route
```

指令执行成功后，会在 `runtime` 目录下面生成 `route.php` 文件，生成的路由缓存文件仅仅支持在应用的路由配置文件中定义的路由（包括方法定义和配置定义）。

清除缓存文件

如果需要清除应用的缓存文件，可以使用下面的命令：

```
php think clear
```

不带任何参数调用clear命令的话，会清除runtime目录（包括模板缓存、日志文件及其子目录）下面的所有的文件，但会保留目录。

如果需要清除某个指定目录下面的文件，可以使用：

```
php think clear --path d:\www\tp5\runtime\log\
```

生成配置缓存文件

可以为应用或者模块生成配置缓存文件

```
php think optimize:config
```

默认生成应用的配置缓存文件，调用后会在 `runtime` 目录下面生成 `init.php` 文件，生成配置缓存文件后，应用目录下面的 `config.php` `common.php` 以及 `tags.php` 不会被加载，被 `runtime/init.php` 取代。

如果需要生成某个模块的配置缓存，可以使用：

```
php think optimize:config --module index
```

调用后会在 `runtime/index` 目录下面生成 `init.php` 文件，生成后，`index` 模块目录下面的 `config.php` `common.php` 以及 `tags.php` 不会被加载，被 `runtime/index/init.php` 取代。

MongoDb

[安装](#)

安装

首先，确保你已经安装了 MongoDB driver for PHP（重要），参考：

<http://pecl.php.net/package/mongodb>

然后使用 Composer 安装 ThinkPHP5.0 的 MongoDB 驱动：

```
composer require topthink/think-mongo
```

修改你的数据库配置文件 `database.php` 中的 `type` 参数为：

```
'type' => '\think\mongo\Connection',
```

接下来可以使用 `Db` 类直接操作 `MongoDb` 了，例如：

```
Db::name('demo')
->find();
Db::name('demo')
->field('id,name')
->limit(10)
->order('id','desc')
->select();
```

或者使用模型操作：

```
User::get(1);
User::all('1,2,3');
```

部署

[虚拟主机环境](#)

[Linux 主机环境](#)

[URL重写](#)

虚拟主机环境

ThinkPHP 支持各种各样的线上生产环境，如果你的生产环境与开发环境不符，需要稍作调整 ThinkPHP 的配置，以适应线上生产环境

修改入口文件

5.0默认的应用入口文件位于 `public/index.php`，内容如下：

```
// 定义应用目录
define('APP_PATH', __DIR__ . '/../application/');
// 加载框架引导文件
require __DIR__ . '/../thinkphp/start.php';
```

入口文件位置的设计是为了让应用部署更安全，`public`目录为web可访问目录，其他的文件都可以放到非WEB访问目录下面。

我们也可以改变入口文件的位置及内容，例如把入口文件改到根目录下面改成：

```
// 应用目录
define('APP_PATH', __DIR__ . '/apps/');
// 加载框架引导文件
require './thinkphp/start.php';
```

注意：APP_PATH的定义支持相对路径和绝对路径，但必须以"/"结束

如果你调整了框架核心目录的位置或者目录名，只需要这样修改：

```
// 改变应用目录的名称
define('APP_PATH', __DIR__ . '/apps/');
// 加载框架引导文件
require './think/start.php';
```

这样最终的应用目录结构如下：

```
www  WEB部署目录（或者子目录）
├─index.php  应用入口文件
├─apps      应用目录
└─think     框架目录
```

Linux 主机环境

部分 Linux 主机设置了 `open_basedir` (可将用户访问文件的活动范围限制在指定的区域, 通常是入口文件根目录的路径) 选项, 导致 ThinkPHP5 访问白屏或者报错

如果把 ThinkPHP5 部署在了 LAMP/LNMP 环境上很有可能出现白屏的情况, 这个时候需要开启 php 错误提示来判断是否是因为设置了 `open_basedir` 选项出错。

打开 `php.ini` 搜索 `display_errors`, 把 Off 修改为 On 就开启了 php 错误提示, 这时再访问之前白屏的页面就会出现错误信息。如果错误信息如下那么很有可能就是因为 `open_basedir` 的问题。

Warning: require(): `open_basedir restriction in effect. File` /home/wwwroot/chunice.com/thinkphp/start.php) is not within the allowed path(s): /index.php on line 21

Warning: require(/home/wwwroot/chunice.com/thinkphp/start.php): failed to open stream: Operation not permitted in /home/wwwroot/chu

Fatal error: require(): Failed opening required '/home/wwwroot/chunice.com/public/../thinkphp/start.php' (include_path='.:usr/local/php/lib/

php.ini 修改方法

把权限作用域由入口文件目录修改为框架根目录

打开 `php.ini` 搜索 `open_basedir`, 把

```
open_basedir = "/home/wwwroot/tp5/public/:/tmp:/var/tmp:/proc/"
```

修改为

```
open_basedir = "/home/wwwroot/tp5/:/tmp:/var/tmp:/proc/"
```

如果你的 `php.ini` 文件的 `open_basedir` 设置选项是被注释的或者为 none, 那么你需要通过 Apache 或者 Nginx 来修改

`php.ini` 文件通常是在 `/usr/local/php/etc` 目录中, 当然了这取决于你 LAMP 环境配置

Apache 修改方法

Apache 需要修改 `httpd.conf` 或者同目录下的 `vhost` 目录下 你的域名.conf 文件, 如果你的生成环境是 LAMP 一键安装包配置那么多半就是直接修改 你的域名.conf 文件

```
apache
├─vhost
│   └─www.thinkphp.cn.conf
│   └─.....
```

```
└─httpd.conf
```

打开 你的域名.conf 文件 搜索 `open_basedir` ,把

```
php_admin_value open_basedir "/home/wwwroot/www.thinkphp.cn/public/:/tmp:/var/tmp:/proc/"
```

修改为

```
php_admin_value open_basedir "/home/wwwroot/www.thinkphp.cn/:/tmp:/var/tmp:/proc/"
```

然后重新启动 `apache` 即可生效

域名.conf 文件通常是在 `/usr/local/apache/conf` 目录中，当然了这取决于你 LAMP 环境配置

Nginx/Tengine 修改方法

Nginx 需要修改 `nginx.conf` 或者 `conf/vhost` 目录下 你的域名.conf 文件，如果你的生成环境是 LNMP/LTMP 一键安装包配置那么多半就是直接修改 你的域名.conf 文件

```
nginx
└─conf
    └─vhost
        └─www.thinkphp.cn.conf
            └─nginx.conf
                └─.....
└─nginx.conf
```

打开 你的域名.conf 文件 搜索 `open_basedir` ,把

```
fastcgi_param PHP_VALUE "open_basedir=/home/wwwroot/www.thinkphp.cn/public/:/tmp:/proc/";
```

修改为

```
fastcgi_param PHP_VALUE "open_basedir=/home/wwwroot/www.thinkphp.cn/:/tmp:/proc/";
```

然后重新启动 Nginx 即可生效

域名.conf 文件通常是在 `/usr/local/nginx/conf/vhost` 目录中，当然了这取决于你 LNMP/LTMP 环境配置

修改 ThinkPHP5 入口文件

直接修改 ThinkPHP5 的入口文件会把你的框架文件及程序目录暴露在外网，敬请注意安全防护。

修改入口文件方法请参考[\(部署-虚拟主机环境\)](#)

URL重写

可以通过URL重写隐藏应用的入口文件 `index.php` ,下面是相关服务器的配置参考：

[Apache]

1. httpd.conf配置文件中加载了mod_rewrite.so模块
2. AllowOverride None 将None改为 All
3. 把下面的内容保存为.htaccess文件放到应用入口文件的同级目录下

```
<IfModule mod_rewrite.c>
Options +FollowSymlinks -Multiviews
RewriteEngine on

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

[IIS]

如果你的服务器环境支持ISAPI_Rewrite的话，可以配置httpd.ini文件，添加下面的内容：

```
RewriteRule (.*)$ /index\.php\?s=$1 [I]
```

在IIS的高版本下面可以配置web.Config，在中间添加rewrite节点：

```
<rewrite>
  <rules>
    <rule name="OrgPage" stopProcessing="true">
      <match url="^(.*)$" />
      <conditions logicalGrouping="MatchAll">
        <add input="{HTTP_HOST}" pattern="^(.*)$" />
        <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
        <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
      </conditions>
      <action type="Rewrite" url="index.php/{R:1}" />
    </rule>
  </rules>
</rewrite>
```

[Nginx]

在Nginx低版本中，是不支持PATHINFO的，但是可以通过在Nginx.conf中配置转发规则实现：

```
location / { // ....省略部分代码
  if (!-e $request_filename) {
    rewrite ^(.*)$ /index.php?s=/$1 last;
    break;
  }
}
```


其实内部是转发到了ThinkPHP提供的兼容URL，利用这种方式，可以解决其他不支持PATHINFO的WEB服务器环境。

如果你的应用安装在二级目录，**Nginx** 的伪静态方法设置如下，其中 **youdomain** 是所在的目录名称。

```
location /youdomain/ {  
    if (!-e $request_filename){  
        rewrite  ^/youdomain/(.*)$  /youdomain/index.php?s=/$1  last;  
    }  
}
```

原来的访问URL：

```
http://serverName/index.php/模块/控制器/操作/[参数名/参数值...]
```

设置后，我们可以采用下面的方式访问：

```
http://serverName/模块/控制器/操作/[参数名/参数值...]
```

附录

[配置参考](#)

[常量参考](#)

[助手函数](#)

[升级指导](#)

[更新日志](#)

配置参考

惯例配置 应用设置

```
// 应用命名空间
'app_namespace'          => 'app',
// 应用调试模式
'app_debug'              => true,
// 应用模式状态
'app_status'             => '',
// 应用Trace
'app_trace'              => false,
// 是否支持多模块
'app_multi_module'       => true,
// 注册的根命名空间
'root_namespace'        => [],
// 扩展配置文件
'extra_config_list'      => ['database', 'route', 'validate'],
// 扩展函数文件
'extra_file_list'        => [THINK_PATH . 'helper' . EXT],
// 默认输出类型
'default_return_type'    => 'html',
// 默认AJAX 数据返回格式,可选json xml ...
'default_ajax_return'    => 'json',
// 默认JSONP格式返回的处理方法
'default_jsonp_handler'  => 'jsonpReturn',
// 默认JSONP处理方法
'var_jsonp_handler'      => 'callback',
// 默认时区
'default_timezone'       => 'PRC',
// 是否开启多语言
'lang_switch_on'         => false,
// 默认全局过滤方法 用逗号分隔多个
'default_filter'         => '',
// 默认语言
'default_lang'           => 'zh-cn',
// 应用类库后缀
'class_suffix'           => false,
// 控制器类后缀
'controller_suffix'     => false,
```

模块设置

```
// 默认模块名
'default_module'         => 'index',
// 禁止访问模块
'deny_module_list'       => ['common'],
// 默认控制器名
'default_controller'     => 'Index',
// 默认操作名
'default_action'         => 'index',
// 默认验证器
'default_validate'       => '',
// 默认的空控制器名
'empty_controller'       => 'Error',
// 操作方法后缀
```

```
'action_suffix'          => '',
// 自动搜索控制器
'controller_auto_search' => false,
```

URL设置

```
// PATHINFO变量名 用于兼容模式
'var_pathinfo'          => 's',
// 兼容PATH_INFO获取
'pathinfo_fetch'        => ['ORIG_PATH_INFO', 'REDIRECT_PATH_INFO', 'REDIRECT_URL'],
// pathinfo分隔符
'pathinfo_depr'         => '/',
// URL伪静态后缀
'url_html_suffix'        => 'html',
// URL普通方式参数 用于自动生成
'url_common_param'       => false,
// URL参数方式 0 按名称成对解析 1 按顺序解析
'url_param_type'         => 0,
// 是否开启路由
'url_route_on'           => true,
// 是否强制使用路由
'url_route_must'         => false,
// 域名部署
'url_domain_deploy'      => false,
// 域名根, 如thinkphp.cn
'url_domain_root'        => '',
// 是否自动转换URL中的控制器和操作名
'url_convert'            => true,
// 默认的访问控制器层
'url_controller_layer'   => 'controller',
// 表单请求类型伪装变量
'var_method'             => '_method',
```

模板引擎设置

```
'template'              => [
    // 模板引擎类型 支持 php think 支持扩展
    'type'                 => 'Think',
    // 模板路径
    'view_path'            => '',
    // 模板后缀
    'view_suffix'          => 'html',
    // 模板文件名分隔符
    'view_depr'            => DS,
    // 模板引擎普通标签开始标记
    'tpl_begin'            => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'              => '}',
    // 标签库标签开始标记
    'taglib_begin'         => '{',
    // 标签库标签结束标记
    'taglib_end'           => '}',
],

// 视图输出字符串内容替换
'view_replace_str'       => [],
// 默认跳转页面所对应的模板文件
'dispatch_success_tmpl'  => THINK_PATH . 'tpl' . DS . 'dispatch_jump.tpl',
'dispatch_error_tmpl'    => THINK_PATH . 'tpl' . DS . 'dispatch_jump.tpl',
```

异常及错误设置

```
// 异常页面的模板文件
'exception_tpl' => THINK_PATH . 'tpl' . DS . 'think_exception.tpl',

// 错误显示信息, 非调试模式有效
'error_message' => '页面错误！请稍后再试~ ',
// 显示错误信息
'show_error_msg' => false,
```

日志设置

```
'log' => [
    // 日志记录方式, 支持 file socket
    'type' => 'File',
    // 日志保存目录
    'path' => LOG_PATH,
],
```

Trace设置

```
'trace' => [
    // 内置Html Console 支持扩展
    'type' => 'Html',
],
```

缓存设置

```
'cache' => [
    // 驱动方式
    'type' => 'File',
    // 缓存保存目录
    'path' => CACHE_PATH,
    // 缓存前缀
    'prefix' => '',
    // 缓存有效期 0表示永久缓存
    'expire' => 0,
],
```

会话设置

```
'session' => [
    'id' => '',
    // SESSION_ID的提交变量, 解决flash上传跨域
    'var_session_id' => '',
    // SESSION 前缀
    'prefix' => 'think',
    // 驱动方式 支持redis memcache memcached
    'type' => '',
    // 是否自动开启 SESSION
    'auto_start' => true,
],
```

Cookie设置

```
'cookie' => [
```

```

// cookie 名称前缀
'prefix'    => '',
// cookie 保存时间
'expire'    => 0,
// cookie 保存路径
'path'      => '/',
// cookie 有效域名
'domain'    => '',
// cookie 启用安全传输
'secure'    => false,
// httponly设置
'httponly'  => '',
// 是否使用 setcookie
'setcookie' => true,
],

```

数据库设置

```

'database'          => [
  // 数据库类型
  'type'             => 'mysql',
  // 数据库连接DSN配置
  'dsn'              => '',
  // 服务器地址
  'hostname'         => 'localhost',
  // 数据库名
  'database'         => '',
  // 数据库用户名
  'username'         => 'root',
  // 数据库密码
  'password'         => '',
  // 数据库连接端口
  'hostport'         => '',
  // 数据库连接参数
  'params'           => [],
  // 数据库编码默认采用utf8
  'charset'          => 'utf8',
  // 数据库表前缀
  'prefix'           => '',
  // 数据库调试模式
  'debug'            => false,
  // 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)
  'deploy'           => 0,
  // 数据库读写是否分离 主从式有效
  'rw_separate'      => false,
  // 读写分离后 主服务器数量
  'master_num'       => 1,
  // 指定从服务器序号
  'slave_no'         => '',
  // 是否严格检查字段是否存在
  'fields_strict'    => true,
  // 数据集返回类型
  'resultset_type'   => 'array',
  // 自动写入时间戳字段
  'auto_timestamp'   => false,
  // 是否需要进行SQL性能分析
  'sql_explain'      => false,
],

```

分页配置

```
'paginate'          => [  
  'type'             => 'bootstrap',  
  'var_page'         => 'page',  
  'list_rows'        => 15,  
],
```

常量参考

预定义常量

预定义常量是指系统内置定义好的常量，不会随着环境的变化而变化，包括：

```
EXT          类库文件后缀（.php）
THINK_VERSION 框架版本号
```

路径常量

系统和应用的路径常量用于系统默认的目录规范，可以通过重新定义改变，如果不希望定制目录，这些常量一般不需要更改。

```
DS 当前系统的目录分隔符
THINK_PATH 框架系统目录
ROOT_PATH 框架应用根目录
APP_PATH 应用目录（默认为application）
CONF_PATH 配置目录（默认为APP_PATH）
LIB_PATH 系统类库目录（默认为 THINK_PATH.'library/'）
CORE_PATH 系统核心类库目录（默认为 LIB_PATH.'think/'）
TRAIT_PATH 系统trait目录（默认为 LIB_PATH.'traits/'）
EXTEND_PATH 扩展类库目录（默认为 ROOT_PATH . 'extend/'）
VENDOR_PATH 第三方类库目录（默认为 ROOT_PATH . 'vendor/'）
RUNTIME_PATH 应用运行时目录（默认为 ROOT_PATH.'runtime/'）
LOG_PATH 应用日志目录（默认为 RUNTIME_PATH.'log/'）
CACHE_PATH 项目模板缓存目录（默认为 RUNTIME_PATH.'cache/'）
TEMP_PATH 应用缓存目录（默认为 RUNTIME_PATH.'temp/'）
```

系统常量

系统常量会随着开发环境的改变或者设置的改变而产生变化。

```
IS_WIN 是否属于Windows 环境
IS_CLI 是否属于命令行模式
THINK_START_TIME 开始运行时间（时间戳）
THINK_START_MEM 开始运行时候的内存占用
ENV_PREFIX 环境变量配置前缀
```


助手函数

系统为一些常用的操作方法封装了助手函数，便于使用，包含如下：

| 助手函数 | 描述 |
|------------|---------------------------------|
| load_trait | 快速导入Traits PHP5.5 以上无需调用 |
| exception | 抛出异常处理 |
| debug | 调试时间和内存占用 |
| lang | 获取语言变量值 |
| config | 获取和设置配置参数 |
| input | 获取输入数据 支持默认值和过滤 |
| widget | 渲染输出Widget |
| model | 实例化Model |
| db | 实例化数据库类 |
| controller | 实例化控制器 |
| validate | 实例化验证器 |
| action | 调用控制器类的操作 |
| import | 导入所需的类库 |
| vendor | 快速导入第三方框架类库 |
| dump | 浏览器友好的变量输出 |
| halt | 变量调试输出并中断执行 |
| url | Url生成 |
| session | Session管理 |
| cookie | Cookie管理 |
| cache | 缓存管理 |
| trace | 记录日志信息 |
| view | 渲染模板输出 |
| request | 实例化Request对象 |
| response | 实例化Response对象 |
| json | JSON数据输出 |
| jsonp | JSONP数据输出 |
| xml | XML数据输出 |
| redirect | 重定向输出 |
| abort | 中断执行并发送HTTP状态码 |
| token | 生成表单令牌输出 |

核心框架不依赖任何助手函数，系统只是默认加载了助手函数，配置如下：

```
// 扩展函数文件定义
'extra_file_list' => [THINK_PATH . 'helper' . EXT],
```

因此，你可以随意修改助手函数的名称或者添加自己的助手函数，然后修改配置为：

```
// 扩展函数文件定义
'extra_file_list' => [APP_PATH . 'helper' . EXT],
```

```
// 使用扩展函数文件
'extra_file_list' => [
    THINK_PATH . 'helper' . EXT,
    APP_PATH . 'helper' . EXT
],
```

升级指导

首先声明本章节并非是指导升级旧的项目到 **5.0**，而是为了使用 **3.x** 版本的开发者更快的熟悉并上手这个全新的版本。同时也强烈建议开发者抛弃之前旧的思维模式，因为 **5.0** 是一个全新的颠覆重构版本。

需要摒弃的3.X旧思想 URL的变动

首先对3.X的不严谨给开发者们带来的不正确的引导表示歉意，在5.0版本正式废除类似/id/1方式 可以通过'get'获取到'id'的方法，严格来讲这样的url是不属于\$_GET的，现在可以通过'param'获取，具体使用可以通过请求部分查询。

模型的变动

新版的模型查询返回默认'对象'，系统默认增加了'toArray'方法，许多开发者在'all'或'select'尝试使用'toArray'来转换为数组，在此希望开发者能理解'对象'的概念，尝试使用'对象'进行数据的使用，或者使用'db'方法进行数据库的操作，也提醒一下部分'滥用'toArray'的开发者，'all'或'select'结果是对象的数组集合，是无法使用'toArray'进行转换的。

新版变化 命名规范

- 目录和文件名采用'小写+下划线'，并且以小写字母开头；
- 类库、函数文件统一以.php为后缀；
- 类的文件名均以命名空间定义，并且命名空间的路径和类库文件所在路径一致（包括大小写）；
- 类名和类文件名保持一致，并统一采用驼峰法命名（首字母大写）

函数

- 系统已经不依赖任何函数，只是对常用的操作封装提供了助手函数；
- 单字母函数废弃，默认系统加载助手函数，具体参考上一个章节'助手函数'；

路由

5.0的URL访问不再支持普通URL模式，路由也不支持正则路由定义，而是全部改为规则路由配合变量规则（正则定义）的方式，具体这里不再赘述。

控制器

控制器的命名空间有所调整，并且可以无需继承任何的控制器类。

- 应用类库的命名空间统一为app（可修改）而不是模块名；
- 控制器的类名默认不带 **Controller** 后缀，可以配置开启 **controller_suffix** 参数启用控制器类后缀；
- 控制器操作方法采用 **return** 方式返回数据，而非直接输出；
- 废除原来的操作前后置方法；

版本对比

3.2版本控制器写法

```
<?php
namespace Home\Controller;

use Think\Controller;

class IndexController extends Controller
{
    public function hello()
    {
        echo 'hello,thinkphp!';
    }
}
```

5.0版本控制器写法

```
namespace app\index\controller;

class Index
{
    public function index()
    {
        return 'hello,thinkphp!';
    }
}
```

3.2版本控制器命名

IndexController.class.php

5.0版本控制器命名

Index.php

怎么才能在控制器中正确的输出模板

5.0在控制器中输出模板，使用方法如下：

如果你继承 `think\Controller` 的话，可以使用：

```
return $this->fetch('index/hello');
```

如果你的控制器没有继承 `think\Controller` 的话，使用：

```
return view('index/hello');
```

模型

如果非要对比与旧版本的改进，模型被分为数据库、模型、验证器三部分，分别对应M方法、模型、自动验证，同时均有所加强，下面做简单介绍。

数据库

5.0的数据库查询功能增强，原先需要通过模型才能使用的链式查询可以直接通过Db类调用，原来的M函数调用可以改用db函数，例如：

3.2版本

```
M('User')->where(['name'=>'thinkphp'])->find();
```

5.0版本

```
db('User')->where('name','thinkphp')->find();
```

模型

新版的模型查询增加了静态方法，例如：

```
User::get(1);
User::all();
User::where('id','>',10)->find();
```

模型部分增强了很多功能，具体请查阅“模型章节”。

自动验证

对比旧的版本，可以理解为之前的自动验证且不同于之前的验证；

ThinkPHP5.0验证使用独立的 `\think\Validate` 类或者**验证器**进行验证，不仅适用于模型，在控制器也可直接调用，具体使用规则请参考“验证”章节，这里不再赘述。

配置文件

新版对配置很多的配置参数或者配置层次都和之前不同了，建议大家要么看看代码，要么仔细通读下官方的开发手册，不要因为配置的问题浪费自己一整天的时间。

异常

5.0对错误零容忍，默认情况下会对任何级别的错误抛出异常，并且重新设计了异常页面，展示了详尽的错误信息，便于调试。

系统常量的废弃

5.0版本相对于之前版本对系统变化进行了大量的废弃，用户如果有相关需求可以自行定义

下面是废除常量

```
REQUEST_METHOD IS_GET IS_POST IS_PUT IS_DELETE IS_AJAX __EXT__ COMMON_MODULE MODULE_NAME
E CONTROLLER_NAME ACTION_NAME APP_NAMESPACE APP_DEBUG MODULE_PATH等
```

部分常量可以在Request里面进行获取，具体参考“请求章节”。

再次说明本章节仅仅为之前使用3.X版本开发者快速理解5.0所写，具体5.0的功能还需要开发者通读手册。

助手函数

5.0 助手函数和 3.2 版本的单字母函数对比如下：

| 3.2 版本 | 5.0 版本 |
|--------|------------|
| C | config |
| E | exception |
| G | debug |
| L | lang |
| T | 废除 |
| I | input |
| N | 废除 |
| D | model |
| M | db |
| A | controller |
| R | action |
| B | 废除 |
| U | url |
| W | widget |
| S | cache |
| F | 废除 |

更新日志

版本更新日志

2016-9-15 正式版

[请求和路由]

- Request对象支持动态绑定属性
- 定义了路由规则的URL原地址禁止访问
- 改进路由规则存储结构
- 路由分组功能增强，支持嵌套和虚拟分组
- 路由URL高效反解
- 改进Request对象param方法获取优先级
- 路由增加name方法设置和获取路由标识
- 增加MISS和AUTO路由规则
- Route类增加auto方法 支持注册一个自动解析URL的路由
- 路由规则支持模型绑定
- 路由变量统一使用param方法获取
- 路由规则标识功能和自动标识
- 增加生成路由缓存指令 optimize:route
- Request对象增加route方法单独获取路由变量
- Request对象的param get post put request delete server cookie env方法的第一个参数传入false 则表示获取原始数据 不进行过滤
- 改进自动路由标识生成 支持不同的路由规则 指向同一个路由标识，改进Url自动生成对路由标识的支持
- 改进Request类 filter属性的初始化
- 改进Request类的isAjax和isPjax方法
- Request类增加token方法
- 路由配置文件支持多个 使用 route_config_file 配置参数配置
- 域名绑定支持https检测
- 改进域名绑定 支持同时绑定模块和其他 支持绑定到数组定义的路由规则，取消域名绑定到分组
- 路由规则增加PATCH请求类型支持
- 增加route_complete_match配置参数设置全局路由规则定义是否采用完整匹配 可以由路由规则的参数 complete_match 进行覆盖
- 改进路由的 后缀参数识别 优先于系统的伪静态后缀参数
- Url类增加root方法用于指定当前root地址（不含域名）
- 改进Url生成对可选参数的支持

[数据库]

- 查询条件自动参数绑定
- 改进分页方法支持参数绑定

- Query类的cache方法增加缓存标签参数
- Query类的update和delete方法支持调用cache方法 会自动清除指定key的缓存 配合查询方法的cache方法一起使用
- 改进Query类的延迟写入方法
- Query类的column和value方法支持fetchsql
- 改进日期查询方法
- 改进存储过程方法exec的支持
- 改进Connection类的getLastInsID方法获取
- 记录数据库的连接日志（连接时间和DSN）
- 改进Query类的select方法的返回结果集判断
- Connection类增加getNumRows方法
- 数据库事务方法取消返回值
- 改进Query类的chunk方法对主键的获取
- 改进当数据库驱动类型使用完整命名空间的时候 Query类的builder方法的问题

[模型]

- 增加软删除功能
- 关联模型和预载入改进
- 关联预载入查询闭包支持更多的连贯操作
- 完善savell方法支持更新和验证
- 关联定义统一返回Relation类
- Model类的has和hasWhere方法对join类型的支持
- Model类的data方法 批量赋值数据的时候 清空原始数据
- Model类的get方法第三个参数传入true的时候会自动更新缓存
- Model类增加只读字段支持
- Model类增加useGlobalScope方法设置是否启用全局查询范围
- Model类的base方法改为静态定义 全局多次调用有效
- Model类支持设定主键、字段信息和字段类型，不依赖自动获取，提高性能
- Model类的data方法 支持修改器
- 改进Relation类对非数字类型主键的支持
- 改进Relation类的一对多删除
- 修正Relation类的一对多关联预载入查询

[日志和缓存]

- 支持日志类型分离存储
- 日志允许设置记录级别
- 增加缓存标签功能
- 缓存类增加pull方法用于获取并删除
- cache助手函数增加tag参数
- 简化日志信息，隐藏数据库密码

- 增加cache/session redis驱动的库选择逻辑;
- memcached驱动的配置参数支持option参数
- 调试模式下面 日志记录增加页面的header和param参数记录
- memcached缓存驱动增加连接账号密码参数
- 缓存支持设置complex类型 支持配置多种缓存并用store切换
- 缓存类增加tag方法 用于缓存标签设置 clear方法支持清除某个缓存标签的数据
- File类型日志驱动支持设置单独文件记录不同的日志级别
- 改进文件缓存和日志的存储文件名命名规范
- 缓存类增加inc和dec方法 针对数值型数据提供自增和自减操作
- Cache类增加has方法 get方法支持默认值

[其它]

- 视图类支持设置模板引擎参数
- 增加表单令牌生成和验证
- 增加中文验证规则
- 增加image和文件相关验证规则
- 重定向Response对象支持with方法隐含传参
- 改进Session类自动初始化
- session类增加pull方法用于获取并删除
- 增加Env类用于获取环境变量
- Request类get/post/put等更改赋值后param方法依然有效
- 改进Jump跳转地址支持Url::build 解析
- 优化Hook类
- 应用调试模式和页面trace支持环境变量设置
- config助手函数支持 config('?name') 用法
- 支持使用BIND_MODULE常量的方式绑定模块
- 入口文件自动绑定模块功能
- 改进验证异常类的错误信息和模板输出, 支持批量验证的错误信息抛出
- 完善console 增加output一些常用的方法
- 增加token助手函数 用于在页面快速显示令牌
- 增加halt方法用于变量调试并中断输出
- 改进Validate类的number验证规则 和 integer区分开
- optimize:autoload增加对extend扩展目录的扫描
- 改进Validate类的boolean验证规则 支持表单数据
- 改进cookie助手函数支持 判断是否存在某个cookie值
- 改进abort助手函数 支持抛出HttpResponseException异常
- 改进File类增加对上传错误的处理
- 改进File类move方法的返回对象增加上传表单信息, 增加获取文件散列值的方法
- 改进File类的move方法的返回对象改为返回File对象实例

- 增加clear和optimize:config 指令
- 改进File类和Validate类的图像文件类型验证
- 控制器的操作方法支持注入Request之外的对象实例
- Request类 param(true) 支持获取带文件的数据
- input助手函数第一个参数增加默认值
- Validate类增加image验证规则 并改进max min length支持多种数据类型
- json输出时数据编码失败后抛出异常

[调整]

- 废除路由映射（静态路由）定义
- 取消url_deny_suffix配置 改由路由的deny_ext参数设置
- 模型save方法返回值改为影响的记录数，取消getId参数
- Request对象controller方法返回驼峰控制器名
- 控制器前置操作方法不存在则抛出异常
- Loader类db方法增加name标识参数
- db助手函数增加第三个参数用于指定连接标识
- Sqlsrv驱动默认不对数据表字段进行小写转换
- 移除sae驱动 改为扩展包
- Oracle驱动移出核心包
- Firebird驱动移出核心包
- 取消别名定义文件alias.php
- 配置参数读取的时候取消环境变量判断 需要读取环境变量时候使用Env类
- 环境变量定义文件更改为 .env 由原来的PHP数组改为ini格式定义（支持数组方式）
- 状态配置和扩展配置的加载顺序调整 便于状态配置文件中可以更改扩展配置参数
- 取消域名绑定到路由分组功能
- 控制器类的success和error方法url参数支持传入空字符串，则不做任何处理
- 控制器的error success result redirect方法均不需要使用return
- 创建目录的权限修改为0644

2016-7-1 RC4版本

[底层架构]

- 增加Request类 并支持自动注入
- 统一Composer的自动加载机制
- 增加Response类的子类扩展
- 增加File类用于上传和文件操作
- 取消模式扩展 SAE支持降权
- 优化框架入口文件
- 改进异常机制
- App类输入/输出调整
- 单元测试的完美支持

- 增加新的控制台指令
- 取消系统路径之外的大部分常量定义
- 类库映射文件由命令行动态生成 包含应用类库

[数据库]

- 增加分表规则方法
- 增加日期和时间表达式查询方法
- 增加分页查询方法
- 增加视图查询方法
- 默认保持数据表字段大小写
- 数据缓存自动更新机制
- 完善事务嵌套支持
- 改进存储过程数据读取
- 支持设置数据库查询数据集返回类型

[模型]

- 增加Merge扩展模型
- 模型支持动态查询
- 增加更多的类型自动转换支持
- 增加全局查询范围
- toJson/toArray支持隐藏和增加属性输出
- 增加远程一对多关联

[其它]

- 日志存储结构调整
- Trace调试功能从日志类独立并增强
- 原Input类功能并入Request类
- 类库映射文件采用命令行生成 包含应用类库
- 验证类的check方法data数据取消引用传参
- 路由增加MISS路由规则
- 路由增加路由别名功能

2016-4-23 RC3版本

[底层架构]

- 框架核心仓库和应用仓库分离 便于composer独立更新
- 数据库类重构，拆分为Connection（连接器）/Query（查询器）/Builder（SQL生成器）
- 模型类重构，更加对象化

[数据库]

- 新的查询语法
- 闭包查询和闭包事务

- Query对象查询
- 数据分批处理
- 数据库SQL执行监听

[模型]

- 对象化操作
- 支持静态调用（查询）
- 支持读取器/修改器
- 时间戳字段
- 对象/数组访问
- JSON序列化
- 事件触发
- 命名范围
- 类型自动转换
- 数据验证和完成
- 关联查询/写入
- 关联预载入

[其它更新]

- 路由类增加快速路由支持
- 验证Validate类重构
- Build类增加快速创建模块的方法
- Url生成类改进
- Validate类改进
- View类及模板引擎驱动设计改进
- 取消模板引擎的模板主题设计
- 修正社区反馈的一些问题
- 助手函数重新命名
- `router.php` 文件位置移动

2016-3-11 RC2版本

- 重新设计的自动验证和自动完成机制（原有自动验证和完成支持采用traits\model\Auto兼容）；
- 验证类Validate独立设计；
- 自动生成功能交给Console完成；
- 对数据表字段大小写的处理；
- 改进Controller类（取消traits\contorller\View）；
- 改进Input类；
- 改进Url类；
- 改进Cookie类；
- 优化Loader类；

- 优化Route类；
- 优化Template类；
- Session类自动初始化；
- 增加traits\model\Bulk模型扩展用于大批量数据写入和更新；
- 缓存类和日志类增加Test驱动；
- 对异常机制和错误处理的改进；
- 增加URL控制器和操作是否自动转换开关；
- 支持类名后缀设置；
- 取消操作绑定到类的功能；
- 取消use_db_switch参数设计；

2016-1-30 RC1版本

[底层架构]

- 真正的惰性加载
- 核心类库组件化
- 框架引导文件
- 完善的类库自动加载（支持Composer）
- 采用Traits扩展
- API友好（输出、异常和调试）
- 文件命名规范调整

[调试和异常]

- 专为API开发而设计的输出、调试和异常处理
- 日志类支持本地文件/SAE/页面Trace/SocketLog输出，可以实现远程浏览器插件调试
- 内置trace方法直接远程调试
- 异常预警通知驱动设计
- 数据库SQL性能分析支持

[路由]

- 动态注册路由
- 自定义路由检测方法
- 路由分组功能
- 规则路由中的变量支持采用正则规则定义（包括全局和局部）
- 闭包路由
- 支持路由到多层控制器

[控制器]

- 控制器类无需继承controller类
- 灵活的多层控制器支持
- 可以Traits引入高级控制器功能
- rest/yar/rpc/hprose/jsonrpc控制器扩展

- 前置操作方法支持排除和指定操作

[模型]

- 简化的核心模型
- Traits引入高级模型/视图模型/关联模型
- 主从分布时候主数据库读操作支持
- 改进的join方法和order方法

[视图]

- 视图解析驱动设计（模板引擎）
- 所有方法不再直接输出而是返回交由系统统一输出处理
- 动态切换模板主题设计
- 动态切换模板引擎设计

[数据库]

- 完全基于PDO实现
- 简化的数据库驱动设计
- SQL性能监控（需要开启数据库调试模式）
- PDO参数绑定改进

[其他方面]

- 目录和MVC文件自动生成支持
- l函数默认添加变量修饰符为/s
- 一个行为类里面支持为多个标签位定义不同的方法
- 更多的社交扩展类库