1-基础

1-基础.note

安装ThinkPHP.note

安装ThinkPHP

> ThinkPHP5的环境要求如下：
>
> - PHP >= 5.4.0
> - PDO 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**），可以通过两种方式获取和安装ThinkPHP。

一、下载ThinkPHP安装

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

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

二、使用Composer安装

ThinkPHP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 topthink/think tp5 dev-master --prefer-dist
```

> 由于目前尚未正式发布，所以先用dev-master分支。

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

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ThinkPHP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

开发规范.note

命名规范

ThinkPHP5的命名规范如下：

目录和文件命名

- 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）

类、方法和属性命名

- 类的命名采用驼峰法，并且首字母大写，例如`User`、`UserType`；
- 方法的命名使用驼峰法，并且首字母小写或者使用下划线“_”，例如`getUserName`，`_parseType`，通常下划线开头的方法属于私有方法；
- 属性的命名使用驼峰法，并且首字母小写或者使用下划线“_”，例如`tableName`、`_instance`，通常下划线开头的属性属于私有属性；
- 以双下划线“__”打头的函数或方法作为魔法方法，例如`__call`和`__autoload`；

函数命名

- 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`；

常量命名

- 常量以大写字母和下划线命名，例如`APP_DEBUG`和`APP_MODE`；

配置命名

- 配置参数以小写字母和下划线命名，例如`url_route_on`；

数据表和字段命名

- 数据表和字段采用小写加下划线方式命名，并注意字段名不要以下划线开头，例如`think_user`表和`user_name`字段，类似`_username`这样的数据表字段可能会被过滤。

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.note

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
│  └─route.php          路由配置文件
│
├─runtime               应用的运行时目录（可写，可设置）
├─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
│  └─.htaccess          用于 apache 的重写
│
├─thinkphp              框架系统目录
│  ├─lang               语言包目录
│  ├─library            框架核心类库目录
│  │  ├─behavior        系统行为类库目录
│  │  ├─think           Think 类库包目录
│  │  └─traits          系统 Traits 目录
│  ├─mode               应用模式目录
│  ├─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
│  ├─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
│
├─vendor                第三方类库目录（Composer）
│
├─.gitignore            Git 忽略规则
├─build.php             自动生成定义文件（参考）
├─composer.json         composer 定义文件
├─console               命令行工具
├─LICENSE.txt           授权说明文件
├─README.md             README 文件
├─router.php            快速测试文件（用于PHP自带webserver）
```

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ThinkPHP5.0.0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。

2-架构.note

2-架构总览.note

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（或者其它应用入口文件）/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`console.php`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基础配置文件、公共文件，也称为应用公共配置和公共函数文件。

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，并且在入口文件中定义如下常量：
>
> ```
> define('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：

```
namespaceapp\index\controller;

classIndex{
    publicfunctionindex(){
        return'hello,thinkphp!';
    }
}
```

操作

一个控制器包含多个操作（方法），操作方法是一个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：

```
namespaceapp\index\controller;

classIndex{
    publicfunctionindex(){
        return'index';
    }

    publicfunctionhello($name){
        return'Hello,'.$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，并且支持模板主题。

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

ThinkPHP采用了PHP的命名空间设计和规划类库文件，并且符合PSR-4的自动加载规范。

入口文件.note

入口文件

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

应该说，所有应用都是从入口文件开始的，并且不同应用的入口文件是类似的。

入口文件定义

入口文件主要完成：

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

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/');
// 开启调试模式
define('APP_DEBUG', true);
// 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。

默认情况下，只需要加载框架入口文件，而不需要调用应用执行方法，系统会自动调用`\think\App`类的run方法，如果你需要在执行run方法之前做其他的操作，可以这样定义入口文件：

```
// 改变应用目录的名称
define('APP_PATH', __DIR__.'/apps/');
// 关闭应用自动执行
define('APP_AUTO_RUN', false);
// 加载框架引导文件require'./think/start.php';
// 这里添加应用执行之前需要的代码// 类已经支持自动加载...// 执行应用
\think\App::run();
```

生命周期.note

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/');
// 开启调试模式
define('APP_DEBUG', true);

// 加载框架引导文件require__DIR__ . '/../thinkphp/start.php';
```

一般入口文件已定义一些常量为主，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；
- 注册自动加载机制；
- 读取模式定义文件；
- 加载模式命名空间定义；
- 加载模式别名定义文件；

- 注册错误和异常处理机制；
- 加载模式配置文件；
- 加载模式行为定义文件；
- 如果开启自动执行则执行应用；

当前应用的模式由常量**APP_MODE**决定，默认为common模式，如果需要更改应用模式则可以在应用入口文件中更改常量定义。

> 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。
系统的自动加载由两个部分组成：

1. 注册系统的自动加载方法 **\think\Loader::autoload**
2. 注册**Composer**自动加载（符合Composer规范即可）

一个类库的自动加载检测顺序为：

1. 是否定义类库映射；
2. Composer自动加载检测；
3. 是否为注册的命名空间；
4. 检测EXTEND_PATH目录下的扩展类库；

如果检测到以上任何一处，则按照**PSR-4**命名规范自动加载。

可以看到，定义类库映射的方式是最高效的。

> 在调试模式下面如果没有加载到会记录一个notic错误。

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；
- 加载扩展配置文件（由extra_config_list定义）；
- 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

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

获取到正常的`$_SERVER['PATH_INFO']`参数后才能继续。

如果有设置`url_deny_suffix`参数，则会过滤非法的URL后缀请求。

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。

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，请不要执行exit中断。

> 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，支持写入到日志文件、页面Trace，甚至是Socket服务器。

URL访问.note

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**，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
```

如果不希望URL自动转换，可以在应用配置文件中设置：

```
// 关闭控制器名的自动转换'url_controller_convert'    =>  false,
// 关闭操作名的自动转换'url_action_convert'        =>  false,
```

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、在应用入口文件添加`.htaccess`文件，内容如下：

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

模块设计.note

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（可更改）
│  ├─runtime            应用的运行时目录（可写，可设置）
│  ├─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         公共配置文件
│  ├─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ThinkPHP5.0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。如果要更改公共模块的名称，可以在入口文件中定义：

```
define('COMMON_MODULE','base');
```

runtime目录是默认的运行时目录，存放应用的相关日志、缓存等。

要更改默认的runtime目录位置，可以在应用入口文件中定义：

```
define('RUNTIME_PATH','./runtime/');
```

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，例如在入口文件中定义：

```
define('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，方法是应用的公共文件中添加如下代码：

```
// 绑定index模块
\think\Route::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...]

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index模块的index控制器
\think\Route::bind('module','index/index');
```

设置后，我们的URL访问地址则变成：

> http://serverName/应用入口/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：

首先在入口文件中定义：

```
// 关闭多模块设计
define('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）
│   ├─runtime           应用的运行时目录（可写，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.note

命名空间

ThinkPHP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ThinkPHP5.0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：

```
namespacethink\cache\driver;
classFile {
}
```

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');
> ```
>
> 请自行使用 use think\Route或者
>
> ```
> \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（`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：

```
namespacemy;
classTest {
    publicfunctionsayHello(){
        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;
classUserextends\think\Model {
}
```

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

```
namespaceapp\admin\Event;
classUser {
}
```

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

自动加载.note

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。

在符合`PSR-4`规范的条件下，类库的自动加载检测顺序如下：

1、优先检测是否定义类库映射
2、检测composer自动加载的类库
3、检测根命名空间是否已经注册

4、检测扩展类库目录EXTEND_PATH下是否存在根命名空间对应的子目录

系统会按顺序检测，一旦检测生效的话，就会按照PSR-4规范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::addMap('think\Log',LIB_PATH.'think\Log.php');
Loader::addMap('org\util\Array',LIB_PATH.'org\util\Array.php');
```

也可以利用addMap方法批量导入类库映射定义，例如：

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

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。

Composer自动加载

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

traits引入.note

ThinkPHP 5.0开始采用PHP5.4的trait功能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。

> Traits 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。Traits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，系统提供了一个快捷的load_trait函数用于自动加载trait类库，例如，可以这样正确引入trait类库。

```
namespaceapp\index\controller;

load_trait('controller/View');  // 引入traits\controller\Viewclassindex{
     use \traits\controller\View;

     publicfunctionindex(){
         $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T函数引入trait。

```
namespaceapp\index\controller;

classindex{
     use \traits\controller\View;

     publicfunctionindex(){

     }
}
```

可以支持同时引入多个trait类库，例如：

```
namespaceapp\index\controller;

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

classindex{
     use \traits\controller\View;
     use \traits\controller\Jump;

     publicfunctionindex(){

     }
}
```

或者使用

```
namespaceapp\index\controller;

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

classindex{
     use \traits\controller\View,\traits\controller\Jump;

     publicfunctionindex(){

     }
}
```

> 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`，是因为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traits的内容可以参考PHP官方手册。

api友好.note

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`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。

```
namespaceapp\index\controller;

classIndex{
    publicfunctionindex(){
        $data = ['name'=>'thinkphp','url'=>'thinkphp.cn'];
        return ['data'=>$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。

核心支持的数据类型包括`html`、`text`、`json`和`jsonp`，其他类型的需要自己扩展，扩展方式为包括两种方式：

第一种方式是调用`Respone::tramsform`方法

```
// 对输出数据设置data_to_xml处理函数（支持callable类型）
\think\Response::tramsform('data_to_xml');
```

第二种方式是对`return_data`钩子使用行为扩展。

```
// 定义行为类
Hook::add('return_data','\app\index\behavior\DataToXml');
// 或者直接使用闭包函数处理
Hook::add('return_data',function(&$data){
    // 在这里对data进行处理
});
```

异常处理

在API开发的情况下，只需要设置好`default_return_type`，就能返回相应格式的异常信息，例如：

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

当发生异常的时候，就会返回客户端一个json格式的异常信息，例如：

```
{"message":"\u53d1\u751f\u9519\u8bef","code":10005}
```

然后由客户端决定如何处理该异常。

错误调试

由于API开发不方便在客户端进行开发调试，但TP5通过扩展日志的输出类型可以支持包括Socket在内的方式，可以实现远程的开发调试。
设置方式：

```
'log'          => [
    '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。

IS_API常量

系统还预留了一个IS_API常量，用于后期API开发的深度优化。

调试模式.note

ThinkPHP5.0增强了开发调整功能，通过强化\think\Log类的功能实现了在不同的环境下面可以使用相同的方法进行调试和输出。

调试方法

系统提供了统一的调试和写入方法，例如：

```
// 记录调试信息到内存 系统由 \think\Log::save()方法统一写入Log::record('调试信息','日志类型');
// 直接写入调试信息Log::write('调试信息','日志类型');
```

最主要的方法就是 `\think\Log::record()`，系统也提供了一个快捷操作方法trace：

```
trace('调试信息','log');
trace(['aaa','bbb'],'debug');
```

> record方法支持各种变量的调试输出。

使用`Log::record()`或者`trace`方法后，通过设置就可以在下面不同的环境中输出调试结果。

本地文件调试

设置如下：

```
'log'=>[
    'type'=>'file',
    'path'=> LOG_PATH,
]
```

会在`LOG_PATH`下面生成调试日志文件。

页面Trace调试

设置如下：

```
'response_exit'=>false,
'log'=>[
    'type'=>'trace',
    'trace_file'=> THNK_PATH.'tpl/page_trace.tpl',
]
```

运行后可以看到右下角出现了如下图示：

点击图标后可以看到详细的页面Trace信息

可以切换显示不同的Tab项查看不同类型的日志信息。

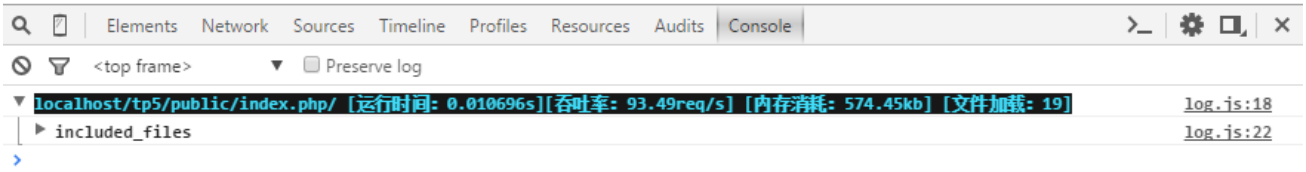
> 需要注意的是，如果要使用页面Trace调试功能，需要设置response_exit为false（否则无法在控制器输出之后显示页面Trace信息）。

SocketLog调试

ThinkPHP5.0版本开始，整合了SocketLog用于本地和远程调试。
设置如下：

```
'log'=>[
    'type'                  => 'socket',
    'host'                  => '111.202.76.133',
    //日志强制记录到配置的client_id'force_client_id'     => '',
    //限制允许读取日志的client_id'allow_client_ids'    => [],
]
```

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：https://chrome.google.com/webstore/detail/socketlog/apkmbfpihjhongonfcgdagliaglghcod （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 &` 我们提供公用的服务端，需要去申请client_id : http://111.202.76.133/

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_id`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_id` 配置项指定要强制输出浏览器的`client_id` 即可。

路由模式.note

ThinkPHP5.0的路由比较灵活，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pathinfo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：

```
module/controller/action/param/value/...
```

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

二、混合模式

开启路由，并使用路由+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，其它的仍然按照默认的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!';
});
```

多层MVC.note

ThinkPHP5.0的路由比较灵活，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pathinfo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：

```
module/controller/action/param/value/...
```

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

二、混合模式

开启路由，并使用路由+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，其它的仍然按照默认的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!';
});
```

自动生成.note

ThinkPHP5.0的路由比较灵活，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pathinfo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：

```
module/controller/action/param/value/...
```

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

二、混合模式

开启路由，并使用路由+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，其它的仍然按照默认的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!';
});
```

console应用.note

ThinkPHP5.0的路由比较灵活，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pathinfo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：

```
module/controller/action/param/value/...
```

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

二、混合模式

开启路由，并使用路由+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，其它的仍然按照默认的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!';
});
```

3--配置.note

配置.note

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、场景配置和动态配置。

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.note

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`yaml/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
URL_MODEL=2 ;URL模式
SESSION_AUTO_START=on ;是否开启session
```

xml格式配置示例：

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

yaml格式配置示例：

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

json格式配置示例：

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

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 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.note

在ThinkPHP中，一般来说应用的配置文件是自动加载的，加载的顺序是：

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

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

下面说明下不同的配置文件的区别和位置：

惯例配置

惯例重于配置是系统遵循的一个重要思想，框架内置有一个惯例配置文件（位于`thinkphp/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，位于`application/config.php`。

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（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。

读取配置.note

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');
```

系统为get方法定义了一个快捷函数`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');
```

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');
```

动态配置.note

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');
// 或者使用快捷方法
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
config(['配置参数1'=>'配置值','配置参数2'=>'配置值']);
```

独立配置.note

独立配置文件

新版支持配置文件分离，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,
],
```

配置作用域.note

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。

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

5-应用.note

应用类库.note

应用类库是指应用的模块下面的各种类库，包括各种控制器类、各种模型类。

方法参数绑定.note

方法参数绑定是通过把URL地址（或者路由地址）中的变量作为操作方法的参数传入。

参数绑定方式是按照变量名进行绑定，例如，我们给Blog控制器定义了两个操作方法read和archive方法，由于read操作需要指定一个id参数，archive方法需要指定年份（year）和月份（month）两个参数，那么我们可以如下定义：

```
namespace app\index\Controller;

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

    public function archive($year='2013',$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/2013/month/11
```

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

> 变量名绑定不一定由访问URL决定，路由地址也能起到相同的作用

输出的结果依次是：

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

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

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

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

如果用户访问的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/Home/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

输入变量.note

概述

系统提供了`\think\Input`类来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Input类来检测一个变量参数是否设置，如下：

```
Input::get('?id');
Input::post('?name');
```

或者使用

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

变量检测可以支持所有Input类支持的系统变量。

变量获取

变量获取使用`\think\Input`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','是否合并全局过滤方法')

变量类型方法包括：

方法	描述
get	获取 $_GET 变量
post	获取 $_POST 变量
put	获取 PUT 变量
param	自动判断请求类型获取 $_GET,$_POST 或者 put 变量
session	获取 $_SESSION 变量
cookie	获取 $_COOKIE 变量
request	获取 $_REQUEST 变量
server	获取 $_SERVER 变量
env	获取 $_ENV 变量
globals	获取 $GLOBALS 变量
path	获取 PATHINFO 变量
file	获取 $_FILES 变量

获取GET变量

```
Input::get('id'); // 获取某个get变量
Input::get('name'); // 获取get变量
Input::get(); // 获取所有的get变量（数组）
```

或者使用内置的快捷I方法实现相同的功能：

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

获取POST变量

```
Input::post('name'); // 获取某个post变量
Input::post(); // 获取全部的post变量
```

使用快捷方法实现：

请求类型.note

判断请求类型

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

系统内置了一些常量用于判断请求类型，包括：

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

使用举例如下：

```
namespace app\index\controller;

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

class User
{
     public function update()
     {
         if (IS_POST)
         {
             $User = new User;
             $User->save(Input::post());
             $this->success('保存完成');
         } else {
             $this->error('非法请求');
         }
     }
}
```

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

伪静态.note

URL伪静态通常是为了满足更好的SEO效果，ThinkPHP支持伪静态URL设置，可以通过设置`url_html_suffix`参数随意在URL的最后增加你想要的静态后缀，而不会影响当前操作的正常执行。例如，我们设置

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

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

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

默认情况下，伪静态的设置为`html`，如果我们设置伪静态后缀为空，

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

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

例如：

```
http://serverName/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，如果要获取当前的伪静态后缀，通过常量 `__EXT__` 获取即可。

如果希望支持多个伪静态后缀，可以直接设置如下：

```
// 多个伪静态后缀设置 用|分割
```

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

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

可以设置禁止访问的URL后缀，例如：

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

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

> 注意：
> url_deny_suffix的优先级比url_html_suffix要高。

6-控制器.note

控制器定义.note

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`类或者其他的控制器类。

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，例如我们在应用的入口文件中定义常量：

```
define('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;

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

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

```php
namespace app\index\controller;

use think\Controller;

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

控制器初始化.note

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：

```php
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.note

可以为某个或者某些操作指定前置执行的操作方法，例如：

```php
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';
    }
}
```

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
```

页面跳转.note

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;

class User extends Controller
{
    public function index()
    {
        $User = new User; //实例化User对象
        $result = $User->save($data);
        if($result){
            //设置成功后跳转页面的地址，默认的返回页面是$_SERVER['HTTP_REFERER']
            return $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
            return $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：

空操作.note

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.note

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;

class Error
{
    public function index()
    {
        //根据当前模块名来判断要执行那个城市的操作
         $cityName = CONTROLLER_NAME;
        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/
```

多级控制器.note

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 $this->show('edit:'.$id);
    }
}
```

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');
```

访问控制器.note

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）。

如果不经过路由访问的话，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

Rest控制器.note

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
{
}
```

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': // put请求处理代码
            break;
        }
    }
}
```

在Rest操作方法中，可以使用$this->_type获取当前访问的资源类型，用$this->_method获取当前的请求类型。
Rest类还提供了response方法用于REST输出：
response输出数据

用法	response($data,$type='',$code=200)
参数	data（必须）：要输出的数据 type（可选）：要输出的类型 支持restOutputType参数允许的类型，如果为空则取restDefaultType参数设置值 code （可选）：HTTP状态
返回值	无

Response方法会自动对data数据进行输出类型编码，目前支持的包括xml json html。
除了普通方式定义Restful操作方法外，系统还支持另外一种自动调用方式，就是根据当前请求类型和资源类型自动调用相关操作方法。系统的自动调用规则是：

RPC控制器.note

7-数据库.note

连接数据库.note

连接数据库.note

> 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、Oracle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'    => '',
];
```

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_LOWER,
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，并且返回数据库的实际列名（默认会转换为小写列名），可以采用下面的方式定义：

```
'params' => [
    \PDO::ATTR_PERSISTENT   => true,
    \PDO::ATTR_CASE         => \PDO::CASE_NATURAL,
],
```

> 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 static $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_',
        // 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'    => '',
    ];
```

```
}
```

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 static $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.note

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。

系统为实例化数据库类提供了一个`db`快捷方法，所以下面的方法是等效的：

```
db($config)->query('select * from think_user where id=:id',['id'=>8]);
```

分布式数据库.note

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

配置`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。

基本查询.note

基本查询.note

基本查询

查询一个数据使用：

```
Db::table('think_user')->where('id',1)->find();
```

查询数据集使用：

```
Db::table('think_user')->where('status',1)->select();
```

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。

添加数据.note

添加数据使用下面的方式：

```
// 指定数据表名称
Db::table('think_user')->insert(
    ['email' => 'thinkphp@qq.com', 'name' => 'thinkphp']
);

// 指定name
Db::name('user')->insert(
    ['email' => 'thinkphp@qq.com', 'name' => 'thinkphp']
);
```

批量添加数据：

```
Db::table('think_user')->insertAll([
    ['email' => 'thinkphp@qq.com', 'name' => 'thinkphp'],
    ['email' => 'admin@qq.com', 'name' => 'admin'],
]);
```

更新数据.note

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]);
```

更新某个字段的值可以使用：

```
Db::table('think_user')
```

```
    ->where('id',1)
    ->setField('name','thinkphp');
```

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);
```

删除数据.note

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();
```

条件查询方法.note

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where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：

表达式查询.note

表达式查询

上面的查询条件仅仅是一个简单的相等判断，可以使用查询表达式支持更多的SQL查询语法，也是ThinkPHP查询语言的精髓，查询表达式的使用格式：

```
where('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

表达式查询的用法示例如下：

EQ：等于（=）

例如：

链式操作.note

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*	用于查询或者更新条件的定义	字符串、数组和对象
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
field	用于定义要查询的字段（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

连贯操作	作用	支持的参数类型
group	用于对查询的group支持	字符串
having	用于对查询的having支持	字符串
join*	用于对查询的join支持	字符串和数组
union*	用于对查询的union支持	字符串、数组和对象
distinct	用于查询的distinct支持	布尔值
lock	用于数据库的锁机制	布尔值
cache	用于查询缓存	支持多个参数
relation	用于关联查询（需要关联模型支持）	字符串
with	用于关联预查询	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
index	用于数据集的强制索引	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.note

---

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();
```

table.note

---

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();
```

alias.note

---

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.note

---

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：

order.note

---

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.note

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）。

page.note

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）。

group.note

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.note

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.note

---

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

union.note

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

distinct.note

DISTINCT 方法用于返回唯一不同的值 。

例如：

```
Db::table('think_user')->distinct(true)->field('name')->select();
```

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

distinct方法的参数是一个布尔值。

lock.note

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.note

cache方法用于查询缓存操作，也是连贯操作方法之一。

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

下面举例说明，例如，我们对find方法使用cache方法如下：

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

```
Db::table('think_user')->cache(true,60,'xcache')->find();
```

表示对查询结果使用xcache缓存，缓存有效期60秒。

cache方法可以指定缓存标识：

```
Db::table('think_user')->cache('key',60)->find();
```

comment.note

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.note

---

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`

index.note

---

index方法用于数据集的强制索引操作，例如：

```
Db::table('think_user')->index('user')->select();
```

对查询强制使用user索引，user必须是数据表实际创建的索引名称。

bind.note

---

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');
```

聚合查询.note

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();
```

或者根据字段统计：

```
Db::table('think_user')->count("id");
```

获取用户的最大积分：

```
Db::table('think_user')->max('score');
```

获取积分大于0的用户的最小积分：

快捷查询.note

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_ti
me` > 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：

子查询.note

可以使用下面两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：

```
// 首先构造子查询SQL
$subQuery = Db::table('think_user')
    ->field('id,name')
    ->where('id','>',10)
    ->select(false);
```

当select方法传入false参数的时候，表示不执行当前查询，而只是生成查询SQL。

2、使用buildSql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。

原生查询.note

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。

可以在query方法中使用表名的简化写法，便于动态更改表前缀，例如：

```
Db::query("select * from __USER__ where status=1");
```

和上面的写法等效，会自动读取当前设置的表前缀。

execute方法

监听sql.note

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：

```php
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。

事务操作.note

使用transaction方法操作数据库事务，当发生异常会自动回滚，例如：

```php
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：

```php
// 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 (\PDOException $e) {
    // 回滚事务
    Db::rollback();
}
```

8-模型.note

模型定义.note

模型定义

定义一个User模型类：

```php
namespace app\index\model;

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

默认主键为id，如果需要指定，可以设置属性：

```php
namespace app\index\model;

class User extends \think\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。

如果你想指定数据表甚至数据库连接的话，可以使用：

```php
namespace app\index\model;
```

```php
class User extends \think\Model
{
    protected static $table = 'think_user';

    protected static $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,
    ];
}
```

新增数据有多种方式。.note

新增数据有多种方式。

实例化方法

第一种是实例化模型对象后赋值并保存：

```php
$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：

```php
$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

```php
$user = new User(['name'=>'thinkphp','email'=>'thinkphp@qq.com']);
$user->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：

```php
$user = new User($_POST);
// post数组中只有name和email字段会写入
$user->field(['name','email'])->save();
```

更新.note

在取出数据后，更改字段内容后更新数据。

```php
$user = User::get(1);
$user->name = 'thinkphp';
$user->save();
```

也可以直接带更新条件来更新数据

```php
$user = new User;
$user->save(['name' => 'thinkphp'],['id' => 1]);
```

可以通过闭包函数使用更复杂的更新条件，例如：

```php
$user = new User;
```

```php
$user->save(['name' => 'thinkphp'],function($query){
    // 更新status值为1 并且id大于10的数据
    $query->where('status',1')->where('id','>',10);
});
```

支持静态方法直接更新数据，例如：

```php
User::where('id',1)->update(['name' => 'thinkphp']);
```

> 如果传入的数据包含主键的话，可以无需使用where方法。

删除.note

、删除当前模型

删除模型数据，可以在实例化后调用`delete`方法。

```php
$user = User::get(1);
$user->delete();
```

根据主键删除

或者直接调用静态方法

```php
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

可以使用闭包函数带条件删除，例如：

```php
User::destroy(function($query){
    $query->where('id','>',10);
});
```

或者通过数据库类的查询条件删除

查询.note

获取单个数据

取出主键为1的数据

```php
$user = User::get(1);
echo $user->name;
// 使用闭包查询
$user = User::get(function($query){
    $query->where('name','thinkphp');
});
echo $user->name;
```

> get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：

```php
// 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;
}
```

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。

聚合.note

在模型中也可以调用数据库的聚合方法进行查询，例如：

```
User::count();
User::where('status','>',0)->count();
User::where('status',1)->avg('score');
User::max('score');
```

数组访问.note

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();
```

json序列化.note

模型对象可以直接被JSON序列化，例如：

```
echo json_encode(User::find(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::find(1);
```

输出的结果和上面是一样的。

获取器.note

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”
```

修改器.note

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);
    }
}
```

可以在修改器中直接修改其他的字段，例如：

```
class User extends Model {
    public function setNameAttr($value,$data){
        $this->data['info'] = $value;
        $this->data['test'] = 'test';
        return $value;
    }
}
```

时间戳.note

如果你定义了自动写入的时间戳字段的话，那么可以无需定义修改器，系统会自动采用时间戳存入数据库，例如在模型类中添加属性定义如下：

```
class User extends Model {
    // 定义需要自动写入时间戳格式的字段
    protected $autoTimeField = ['create_time','update_time'];
    // 以上定义需要配合insert、update或者auto才能生效
```

```
    protected $insert = ['create_time'];
    protected $update = ['update_time'];
}
```

然后，使用下面代码的时候，系统会自动写入`create_time`字段。

```
$user = new User();
$user->name = 'THINKPHP';
$user->save();
echo $user->create_time; // 输出类似 14601053899
```

系统默认设置了三个自动写入的时间戳字段，分别为`create_time`、 `update_time` 和 `delete_time`，实际使用的时候，你只需要加入insert或者update、auto三个属性中即可完成自动写入，或者更改为自己的实际字段名。

```
class User extends Model {
    protected $autoTimeField = ['create_on','update_on'];
    protected $insert = ['create_on'];
    protected $update = ['update_on'];
}
```

类型转换.note

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（整形）后，该字段写入和输出的时候都会自动转换为整形。

`float`

该字段的值写入和输出的时候自动转换为浮点型。

`boolean`

数据完成.note

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，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 = [ 'update_time'];
    protected $insert = [
        'create_time','name','status'=>1
    ];
    protected $update = ['name'];

    protected function setNameAttr($value){
        return strtolower($value);
    }
}
```

在新增数据的时候，会对`create_time`、`update_time`、`name`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->create_time; // 14601053899
```

在保存操作的时候，会自动写入`update_time`字段的值。

```
$user = User::find(1);
$user->name = 'THINKPHP';
$user->save();
echo $user->name; // thinkphp
echo $user->create_time; // 14601053908
```

命名范围.note

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{

    public function scopeThinkphp($query){
        $query->where('name','thinkphp')->field('id,name');
    }

    public 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();
```

模型分层.note

ThinkPHP支持模型的分层，除了Model层之外，我们可以项目的需要设计和创建其他的模型层。

通常情况下，不同的分层模型仍然是继承系统的`\think\Model`类或其子类，所以，其基本操作和`Model`类的操作是一致的。

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

- 数据层：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;
class User extends \think\Model
{
}
```

实例化方法：`\think\Loader::model('User')`

Logic类：app\index\logic\User.php

```
namespace app\index\logic;
class User extends \think\Model
{
}
```

事件.note

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;
    }
});
User::event('before_insert',[$this,'beforeInsert']);
```

一对一关联.note

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');
```

```
    }
}
```

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');
    }
}
```

根据关联查询条件查询

一对多关联.note

一对多关联

关联定义

一对多关联的情况也比较常见，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

```
$article = Article::find(1);
// 获取文章的所有评论
dump($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

多对多关联.note

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');
    }
}
```

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

动态属性.note

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.note

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');
```

聚合模型.note

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 static $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',
    ];
}
```

9-视图.note

视图实例化.note

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'     => DS,
]);
 // 渲染模板输出 并赋值模板变量
return $view->fetch('hello',['name'=>'thinkphp']);
```

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']);
```

模板引擎.note

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。有两种方式进行模板引擎的初始化。

实例化视图的时候初始化

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
    'compile_type'   => 'file', // 模板编译类型
]);
```

> 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();
$View->engine('php')->fetch();
```

表示当前视图的模板文件使用原生php进行解析。

模板赋值.note

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

```
// 实例化视图类
$view = new View();
$view->assign('name','ThinkPHP');
$view->assign('email','thinkphp@qq.com');
 // 或者批量赋值
$view->assign([
    'name'=>'ThinkPHP',
    'email'=>'thinkphp@qq.com'
    ]);
 // 模板输出
return $view->fetch('index');
```

对象赋值

```
// 实例化视图类
$view = new View();
$view->name =   'ThinkPHP';
$view->email=   'thinkphp@qq.com';
 // 模板输出
return $view->fetch('index');
```

传入fetch方法

可以在渲染模板或者内容的时候直接传入模板变量，例如：

```
return $view->fetch('index',[
    'name'=>'ThinkPHP',
    'email'=>'thinkphp@qq.com'
    ]);
```

模板渲染.note

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：

如果当前没有启用模板主题则定位到：

```
当前模块/默认视图目录/当前控制器（小写）/当前操作（小写）.html
```

模板输出替换.note

模板输出替换

支持对视图输出的内容进行字符替换，例如：

```
$view = new View();
$view->replace(['__PUBLIC__'=>'/public/'])->fetch();
```

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__' => '/',
]
```

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助手方法渲染输出。

10-模板.note

模板.note

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。

模板文件定义.note

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();
```

模板标签.note

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'    =>  '>',
],
```

变量输出.note

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';
$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}!

系统变量输出.note

系统变量输出

普通的模板变量需要首先赋值后才能在模板中输出，但是系统变量则不需要，可以直接在模板中输出，系统变量的输出通常以**{$Think** 打头，例如：

```
{$Think.server.script_name} // 输出$_SERVER['SCRIPT_NAME']变量
{$Think.session.user_id} // 输出$_SESSION['user_id']变量
{$Think.get.pageNumber} // 输出$_GET['pageNumber']变量
```

```
{$Think.cookie.name}  // 输出$_COOKIE['name']变量
```

支持输出 $_SERVER、$_ENV、 $_POST、 $_GET、 $_REQUEST、$_SESSION和 $_COOKIE变量。

常量输出

还可以输出常量

```
{$Think.const.MODULE_NAME}
```

或者直接使用

```
{$Think.MODULE_NAME}
```

配置输出

使用函数.note

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}
```

使用默认值.note

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.note

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}

运算符	使用示例
*	{$a*$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.note

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==、===、!=、!==、>=、<=

团样输出.note

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。

模板注释.note

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.note

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。

模板继承.note

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>
```

包含文件.note

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@主题:控制器/操作**

例如：

```
{include file="public/header" /} // 包含头部模版header
{include file="public/menu" /} // 包含菜单模版menu
{include file="blue:public/menu" /} // 包含blue主题下面的menu模版
```

可以一次包含多个模版，例如：

```
{include file="public/header,public/menu" /}
```

> 注意，包含模版文件并不会自动调用控制器的方法，也就是说包含的其他模版文件中的变量赋值需要在当前操作中完成。

标签库.note

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：

内置标签.note

内置标签

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

内置标签包括：

标签名	作用	包含属性
include	包含外部模板文件（闭合）	file
import	导入资源文件（闭合 包括js css load别名）	file,href,type,value,basepath
volist	循环数组数据输出	name,id,offset,length,key,mod
foreach	数组或对象遍历输出	name,item,key
for	For循环数据输出	name,from,to,before,step
switch	分支判断输出	name
case	分支判断输出（必须和switch配套使用）	value,break
default	默认情况输出（闭合 必须和switch配套使用）	无
compare	比较输出（包括eq neq lt gt egt elt heq nheq等别名）	name,value,type
range	范围判断输出（包括in notin between notbetween别名）	name,value,type
present	判断是否赋值	name
notpresent	判断是否尚未赋值	name
empty	判断数据是否为空	name
notempty	判断数据是否不为空	name
defined	判断常量是否定义	name
notdefined	判断常量是否未定义	name
define	常量定义（闭合）	name,value
assign	变量赋值（闭合）	name,value
if	条件判断输出	condition
elseif	条件判断输出（闭合 必须和if标签配套使用）	condition
else	条件不成立输出（闭合 可用于其他标签）	无

标签名	作用	包含属性
php	使用php代码	无

循环输出标签.note

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

循环输出标签(1).note

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

条件判断.note

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。

条件判断(1).note

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。

资源文件加载.note

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='/Public/Js/Util/Array.js'>
<link rel="stylesheet" type="text/css" href="/App/Tpl/default/Public/css/style.css" />
```

系统提供了专门的标签来简化上面的导入：

第一个是import标签，导入方式采用类似ThinkPHP的import函数的命名空间方式，例如：

```
{import type='js' file="Js.Util.Array" /}
```

Type属性默认是js，所以下面的效果是相同的：

```
{import file="Js.Util.Array" /}
```

还可以支持多个文件批量导入，例如：

```
{import file="Js.Util.Array,Js.Util.Date" /}
```

导入外部CSS文件必须指定type属性的值，例如：

资源文件加载(1).note

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='/Public/Js/Util/Array.js'>
<link rel="stylesheet" type="text/css" href="/App/Tpl/default/Public/css/style.css" />
```

系统提供了专门的标签来简化上面的导入：

第一个是import标签，导入方式采用类似ThinkPHP的import函数的命名空间方式，例如：

```
{import type='js' file="Js.Util.Array" /}
```

Type属性默认是js，所以下面的效果是相同的：

```
{import file="Js.Util.Array" /}
```

还可以支持多个文件批量导入，例如：

```
{import file="Js.Util.Array,Js.Util.Date" /}
```

导入外部CSS文件必须指定type属性的值，例如：

标签嵌套.note

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.note

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

定义标签.note

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：

11-日志.note

介绍.note

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。

记录方式.note

记录方式

记录方式

日志的记录方式默认是文件方式，可以通过驱动的方式来扩展支持更多的记录方式。

记录方式由`log.type`参数配置，例如：

```
'log'   => [
    'type' => 'File',
    'path' => LOG_PATH,
],
```

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`。

> 不同的日志驱动可能有不同的参数配置。

目前支持的记录方式包含下面几种：

文件方式

日志信息记录在本地的文件中，适合所有的开发，配置参数包括：

参数名	描述
path	日志文件的保存路径，默认使用 LOG_PATH
file_size	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
time_format	日志的时间格式，默认是`c`

日志写入.note

日志写入

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，系统已经支持的写入方式包括File、Sae、Socket，如果要使用Socket方式写入，可以设置：

```
'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'      => [],
],
```

Socket驱动采用了SocketLog类库，更多的关于SocketLog的资料请参考调试章节。

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',
],
```

手动记录.note

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');
```

日志通知.note

日志通知

异常通知

当发生异常的情况下，可以设置是否发送通知，要启用该功能，首先需要设置通知参数，在应用的公共文件添加如下代码：

```
// 设置异常通知方式 采用邮件发送
Log::alarm(['type'=>'email','address'=>'thinkphp@q.com']);
```

可以通过添加驱动支持其他的通知方式。

日志清空.note

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();
```

12-错误和调试.note

调试模式.note

调试模式

ThinkPHP有专门为开发过程而设置的调试模式，开启调试模式后，会牺牲一定的执行效率，但带来的方便和除错功能非常值得。

> 我们强烈建议ThinkPHP开发人员在开发阶段始终开启调试模式（直到正式部署后关闭调试模式），方便及时发现隐患问题和分析、解决问题。

开启调试模式很简单，只需要在入口文件中增加一行常量定义代码：

```
// 开启调试模式
define('APP_DEBUG', true);
```

在完成开发阶段部署到生产环境后，只需要关闭调试模式或者删除调试模式定义代码即可切换到部署模式。

```
// 关闭调试模式
define('APP_DEBUG', false);
```

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；
- 关闭字段缓存，数据表字段修改不受缓存影响；

- 严格检查文件大小写（即使是Windows平台），帮助你提前发现Linux部署可能导致的隐患问题；
- 通过页面Trace功能更好的调试和发现错误；
- 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.note

异常处理

和PHP默认的异常处理不同，ThinkPHP抛出的不是单纯的错误信息，而是一个人性化的错误页面，如下图所示：

[10005] Exception in App.php line 204

module [hello] not exists

```
195.            define('MODULE_NAME', strip_tags($module));
196.
197.            // 模块初始化
198.            if (MODULE_NAME && !in_array(MODULE_NAME, $config['deny_module_list']) && is_dir(APP_PATH . MODULE_NAME)) {
199.                define('MODULE_PATH', APP_PATH . MODULE_NAME . DS);
200.                define('VIEW_PATH', MODULE_PATH . VIEW_LAYER . DS);
201.                // 初始化模块
202.                self::initModule(MODULE_NAME, $config);
203.            } else {
204.                throw new Exception('module [ ' . MODULE_NAME . ' ] not exists ', 10005);
205.            }
206.        } else {
207.            // 单一模块部署
208.            define('MODULE_NAME', '');
209.            define('MODULE_PATH', APP_PATH);
210.            define('VIEW_PATH', MODULE_PATH . VIEW_LAYER . DS);
211.        }
212.
213.        // 获取控制器名
```

Call Stack

1. in App.php line 204
2. at App::module(['hello', *null*, *null*], ['app_status' => '', 'root_namespace' => [], 'extra_config_list' => ['database', 'route'], ...]) in App.php line 86
3. at App::run() in start.php line 50
4. at require('D:\www\tp5\thinkphp\...') in index.php line 20

Environment Variables

GET Data empty

POST Data empty

Files empty

Cookies

thinkphp_show_page_trace	0|3

Session empty

Server/Request Data

REDIRECT_STATUS	200
HTTP_ACCEPT	text/html, application/xhtml+xml, image/jxr, */*
HTTP_ACCEPT_LANGUAGE	zh-Hans-CN,zh-Hans;q=0.5

> 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，可以做如下设置，忽略某种类型的错误：

```
// 异常处理忽略的错误类型，支持PHP所有的错误级别常量，多个级别可以用|运算法
// 参考：http://php.net/manual/en/errorfunc.constants.php
'exception_ignore_type'   => E_WARNING|E_USER_WARNING|E_NOTICE|E_USER_NOTICE,
```

设置后，`E_WARNING|E_USER_WARNING|E_NOTICE|E_USER_NOTICE`级别的错误就不会抛出异常了。

只有在调试模式下面才能显示具体的错误信息，如果在部署模式下面，你可能看到的是一个简单的提示文字，例如：

一旦关闭调试模式，发生错误后不会提示具体的错误信息，如果你仍然希望看到具体的错误信息，那么可以如下设置：

```
'show_error_msg'        =>  true,    // 显示错误信息
```

另外一种方式是配置**error_page**参数，把所有异常和错误都重定向到一个统一页面，从而避免让用户看到异常信息，通常在部署模式下面使用。`error_page`参数必须是一个完整的URL地址，例如：

```
'error_page' =>'/public/error.html'
```

如果不在当前域名，还可以指定域名：

```
'ERROR_PAGE' =>'http://www.myDomain.com/public/error.html'
```

页面Trace.note

页面Trace

调试模式并不能完全满足我们调试的需要，有时候我们需要手动的输出一些调试信息。除了本身可以借助一些开发工具进行调试外，ThinkPHP还提供了一些内置的调试工具和函数。例如，页面Trace功能就是ThinkPHP提供给开发人员的一个用于开发调试的辅助工具。可以实时显示当前页面的操作的请求信息、运行情况、SQL执行、错误提示等，并支持自定义显示。

页面Trace功能对调试模式和部署模式都有效，不过只能用于有页面输出的情况（如果你的操作没有任何输出，那么可能页面Trace功能对你帮助不大，你可能需要使用其它的调试方法）。

> 在部署模式下面，显示的调试信息没有调试模式完整，通常我们建议页面Trace配合调试模式一起使用。

要开启页面Trace功能，需要配置日志记录方式为trace即可：

```
// 显示页面Trace信息
'log' =>[
    'type'  =>  'trace',
]
```

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，会展开详细的页面Trace信息，如图：

页面Trace框架有6个选项卡，分别是基本、文件、流程、错误、SQL和调试，点击不同的选项卡会切换到不同的Trace信息窗口。

选项卡	描述
基本	当前页面的基本摘要信息，例如执行时间、内存开销、文件加载数、查询次数等等
文件	详细列出当前页面执行过程中加载的文件及其大小
流程	会列出当前页面执行到的行为和相关流程
错误	当前页面执行过程中的一些错误信息，包括警告错误
SQL	当前页面执行到的SQL语句信息
调试	开发人员在程序中进行的调试输出

页面Trace的选项卡是可以定制和扩展的，默认的配置为：

```
// 显示页面Trace信息
'log' =>[
    'type'       =>  'trace',
    '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：

```
// 显示页面Trace信息
'log' =>[
    'type'       =>  'trace',
    '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：

```
// 显示页面Trace信息
'log' =>[
    'type'       =>  'trace',
    'trace_tabs' =>  [
         'base'=>'基本',
         'file'=>'文件',
         'error|notice'=>'错误',
         'sql'=>'SQL',
         'debug|log|info'=>'调试',
     ]
]
```

更改后的页面Trace显示效果如图：

变量调试.note

变量调试

除了页面Trace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) ""
}
```

性能调试.note

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。

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.note

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' => NU
LL, '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页面Trace或者日志文件。

13-验证.note

验证器.note

验证器

概述

ThinkPHP5.0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());
}
```

> 需要注意的是，由于使用了引用传值，check方法的data参数必须传入变量。

如果你是在控制器或者模型类中进行验证，可以使用系统提供的validate方法更简单的进行验证，详细参考后续章节。

验证规则.note

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：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
];
$validate = new Validate($rule);
```

也可以使用rule方法动态添加规则，例如：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
];
$validate = new \Validate($rule);
$validate->rule('zip', '/^\d{6}$/');
$validate->rule([
    'email'   => 'email',
]);
```

规则定义

规则定义支持下面两种方式：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
];
$validate = new Validate($rule);
```

或者

```
$rule = [
    'name'  => ['require','max'=>25'],
    'age'   => ['number','between'=>'1,120'],
];
$validate = new Validate($rule);
```

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 extend 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

错误信息.note

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.note

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 extend 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：

```php
$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.note

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：

```php
$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：

```php
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：

```php
$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：

```
$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.note

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.note

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'
```

max:number

验证某个字段的值的最大长度，例如：

```
'name'=>'max:25'
```

min:number

验证某个字段的值的最小长度，例如：

```
'name'=>'min:5'
```

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'
```

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：

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

验证某个字段的值等于某个值的时候必须，例如：

```
// 当account有值的时候password字段必须
'password'=>'requireWith:account'
```

14-缓存.note

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。

```
$options = [
    'type'=>'File', // 缓存类型为File
    'expire'=>0, // 缓存有效期为永久有效
    'prefix'=>'think'
    'path'=> APP_PATH.'Runtime/cache/', // 指定缓存目录
];
Cache::connect($options);
```

或者通过定义配置参数的方式：

```
'cache'  => [
    'type'   => 'File',
    'path'   => CACHE_PATH,
    'prefix' => '',
    'expire' => 0,
],
```

支持的缓存类型包括file、apachenote、apc、eaccelerator、memcache、secache、wincache、shmop、sqlite、db、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）
length	缓存队列长度（默认为0）

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

```
Cache::set('name',$value,3600); // 缓存标识为name的数据，有效期3600秒
```

获取缓存

获取缓存数据可以使用：

```
dump(Cache::get('name')); // 获取缓存数据
```

删除缓存

```
Cache::rm('name'); // 删除name标识的缓存数据
```

清空缓存

```
Cache::clear(); // 清空缓存数据
```

缓存队列

所有缓存方式均支持缓存队列功能，也就是说，我们可以指定缓存队列的长度，超出长度后，位于队列开头的缓存数据将会被移除。

我们只需要设置length属性，例如：

```
$options = [
    'type'  =>  'File', // 缓存类型为File
    'expire'=>  0, // 缓存有效期为永久有效
    'length'=>  3, // 缓存队列长度
    'temp'  =>  APP_PATH.'Runtime/Cache/', // 指定缓存目录
];
Cache::connect($options);
Cache::set('name1','val1');
Cache::set('name2','val2');
Cache::set('name3','val3');
Cache::set('name4','val4'); // name1标识的缓存数据将被移除
dump(Cache::get('name1')); // 输出结果为 false
```

快捷方法

系统对缓存操作提供了快捷函数`cache`，用法如下：

```
$options = [
    'type'  =>  'File', // 缓存类型为File
    'expire'=>  0, // 缓存有效期为永久有效
    'length'=>  3, // 缓存队列长度
    'path'  =>  APP_PATH.'Runtime/cache/', // 指定缓存目录
 ];
cache($options); // 缓存初始化
cache('name', $value, 3600); // 设置缓存数据
var_dump(cache('name')); // 获取缓存数据
cache('name', NULL); // 删除缓存数据
```

还可以在设置缓存的同时进行参数设置：

```
cache('test',$value,['expire'=>60,'path'=>APP_PATH.'Temp/','type'=>'xcache']);
```

15-Session.note

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。

基础用法

```
// 赋值（当前作用域）
Session::set('name','thinkphp');
// 赋值think作用域
Session::set('name','thinkphp','think');
// 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（当前作用域）
Session::get('name');
// 取值think作用域
Session::get('name','think');
// 指定当前作用域
Session::prefix('think');
// 删除（当前作用域）
Session::delete('name');
// 删除think作用域下面的值
Session::delete('name','think');
// 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，配置如下：

```
'session'               => [
    'prefix'         => 'module',
    'type'           => 'redis',
    'auto_start'     => true,
    'host'          => '127.0.0.1',  // redis主机
    'port'          => 6379,         // redis端口
    'password'      => '',           // 密码
],
```

表示使用redis作为session类型。

16-Cookie.note

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]);
```

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：

```php
// 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_');
```

17-多语言.note

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：

```php
'lang_switch_on' => true,   // 开启语言包功能
'lang_list'      => ['zh-cn'], // 支持的语言列表
```

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。

```php
// 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：

```php
Lang::get('add user error');
// 使用系统封装的快捷方法
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：

```php
Lang::load(APP_PATH . 'common\lang\zh-cn.php');
```

ThinkPHP语言文件定义采用返回数组方式：

```php
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：

```
{:L('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：

```
{:L('file_format',['jpeg,png,gif,jpg','2MB'])}
```

18-扩展.note

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。

ThinkPHP5.0增加了一个行为的总开关常量，要使用行为必须先开启，如下：

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

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
path_info	PATH_INFO检测标签位
app_begin	应用开始标签位
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
error_output	异常信息输出标签（仅对非html输出类型有效）

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：

```
Hook::add('app_init','behavior\\CheckLang'); // 注册 behavior\CheckLang行为类到app_init标签位
Hook::add('app_init','admin\\behavior\\CronRun');//注册 admin\behavior\CronRun行为类到app_init标签位
```

如果要批量注册行为的话，可以使用：

```
Hook::add('app_init',['behavior\\CheckAuth','behavior\\CheckLang','admin\\behavior\\CronRun']);
```

当应用运行到`app_init`标签位的时候，就会依次调用`behavior\CheckAuth`、`behavior\CheckLang`和`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'=> [
        'behavior\\CheckAuth',
        'behavior\\CheckLang'
    ],
    'app_end'=> [
        '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'=> [
        'behavior\\CheckAuth',
        '_overlay'=>true
    ],
    'app_end'=> [
        '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 behavior\CheckAuth行为类的run方法 并引用传入params参数
$result = Hook::exec('behavior\\CheckAuth','run',$params);
```

19-部署.note

虚拟主机环境.note

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/');
// 开启调试模式
define('APP_DEBUG', true);
// 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/');
// 开启调试模式
define('APP_DEBUG', true);
// 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.note

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 oepn_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 oepn_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/p

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)**

20-配置参考.note

配置参考.note

配置参考

惯例配置

应用设置

```
'app_status'             => '',    // 应用模式状态
'root_namespace'         => [],    // 注册的根命名空间
'extra_config_list'      => ['database', 'route', 'validate', 'auto'],  // 扩展配置文件
'extra_file_list'        => [THINK_PATH . 'helper' . EXT],  // 扩展函数文件
'default_return_type'    => 'html',  // 默认输出类型
'default_lang'           => 'zh-cn',  // 默认语言
'response_return'        => false,  // response是否返回方式
'default_ajax_return'    => 'JSON',  // 默认AJAX 数据返回格式,可选JSON XML ...
'default_jsonp_handler'  => 'jsonpReturn',  // 默认JSONP格式返回的处理方法
'var_jsonp_handler'      => 'callback',  // 默认JSONP处理方法
'default_timezone'       => 'PRC',  // 默认区时
'lang_switch_on'         => false,  // 是否开启多语言
```

```
'default_filter'          => '',  // 默认全局过滤方法 用逗号分隔多个
'response_auto_output'    => true,  // 自动Response输出
```

模块设置

```
'default_module'          => 'index',  // 默认模块名
'deny_module_list'        => [COMMON_MODULE, 'runtime'],  // 禁止访问模块
'default_controller'      => 'Index',  // 默认控制器名
'default_action'          => 'index',  // 默认操作名
'empty_controller'        => 'Error',  // 默认的空控制器名
'action_suffix'           => '',  // 操作方法后缀
```

URL 设置

```
'var_pathinfo'            => 's',  // PATHINFO变量名 用于兼容模式
'pathinfo_fetch'          => ['ORIG_PATH_INFO', 'REDIRECT_PATH_INFO', 'REDIRECT_URL'],  // 兼容PATH_INFO获
取
'pathinfo_depr'           => '/',  // pathinfo分隔符
'url_request_uri'         => 'REQUEST_URI',  // 获取当前页面地址的系统变量 默认为REQUEST_URI
'base_url'                => $_SERVER["SCRIPT_NAME"],  // 基础URL路径
'url_html_suffix'         => '.html',  // URL伪静态后缀
'url_common_param'        => false,  // URL普通方式参数 用于自动生成
'url_deny_suffix'         => 'ico|png|gif|jpg',  //url禁止访问的后缀
'url_route_on'            => true,  // 是否开启路由
'url_route_must'          => false,  // 是否强制使用路由
'url_module_map'          => [],  // URL模块映射
'url_domain_deploy'       => false,  // 域名部署
'url_domain_root'         => '',  // 域名根，如.thinkphp.cn
'url_controller_convert' => true,  // 是否自动转换URL中的控制器名
'url_action_convert'      => true,  // 是否自动转换URL中的操作名
```

模板设置

```
'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

```
'exception_tmpl'          => THINK_PATH . 'tpl' . DS . 'think_exception.tpl',  // 异常页面的模板文件
'exception_ignore_type'   => 0, // 异常处理忽略的错误类型，支持PHP所有的错误级别常量，多个级别可以用|运算法。参考：h
ttp://php.net/manual/en/errorfunc.constants.php
'error_message'           => '页面错误！请稍后再试～',  // 错误显示信息,非调试模式有效
'error_page'              => '',  // 错误定向页面
'show_error_msg'          => false,  // 显示错误信息
```

日志设置

```
'log'                     => [
    'type' => 'File',  // 日志记录方式，支持 file socket trace sae
    'path' => LOG_PATH,  // 日志保存目录
```

```
],
```

缓存设置

```
'cache'                 => [
    'type'   => 'File',  // 驱动方式
    'path'   => CACHE_PATH,  // 缓存保存目录
    'prefix' => '',  // 缓存前缀
    'expire' => 0,  // 缓存有效期 0表示永久缓存
],
```

SESSION 设置

```
'session'               => [
    'id'             => '',
    'var_session_id' => '',  // SESSION_ID的提交变量,解决flash上传跨域
    'prefix'         => 'think',   // session 前缀
    'type'           => '',  // 驱动方式 支持redis memcache memcached
    'auto_start'     => true,  // 是否自动开启 session
],
```

数据库设置

```
'database'              => [
    'type'        => 'mysql',  // 数据库类型
    'dsn'         => '',  // 数据库连接DSN配置
    'hostname'    => 'localhost',  // 服务器地址
    'database'    => '',  // 数据库名
    'username'    => 'root',  // 数据库用户名
    'password'    => '',  // 数据库密码
    'hostport'    => '',  // 数据库连接端口
    'params'      => [],  // 数据库连接参数
    'charset'     => 'utf8',  // 数据库编码默认采用utf8
    'prefix'      => '',  // 数据库表前缀
    'debug'       => false,  // 数据库调试模式
    'deploy'      => 0,  // 数据库部署方式:0 集中式(单一服务器),1 分布式(主从服务器)
    'rw_separate' => false,  // 数据库读写是否分离 主从式有效
    'master_num'  => 1,  // 读写分离后 主服务器数量
    'slave_no'    => '',  // 指定从服务器序号
],
```

常量参考.note

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）
LIB_PATH 系统类库目录（默认为 THINK_PATH.'library/'）
CORE_PATH 系统核心类库目录 （默认为 LIB_PATH.'think/'）
MODE_PATH 系统应用模式目录 （默认为 THINK_PATH.'mode/'）
TRAIT_PATH 系统trait目录（默认为 LIB_PATH.'traits/'）
COMMON_MODULE 公共模块名称 （默认为 'common'）
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 APP_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。

```
MODEL_LAYER 默认模型层的名称
CONTROLLER_LAYER 默认控制器层的名称
VIEW_LAYER 默认视图层的名称
APP_NAMESPACE 应用的根命名空间名称
APP_AUTO_RUN 是否自动执行应用
APP_ROUTE_ON 是否允许路由
APP_ROUTE_MUST 是否强制使用路由
IS_API 是否属于API模式
IS_CGI 是否属于CGI模式
IS_WIN 是否属于Windows 环境
IS_CLI 是否属于命令行模式
__INFO__ 当前的PATH_INFO字符串
__EXT__ 当前URL地址的扩展名
MODULE_NAME 当前模块名
MODULE_PATH 当前模块路径
CONTROLLER_NAME 当前控制器名
ACTION_NAME 当前操作名
APP_MULTI_MODULE 多模块支持
APP_DEBUG 是否开启调试模式
APP_HOOK 是否开启Hook
APP_MODE 当前应用模式名称
START_TIME 开始运行时间（时间戳）
NOW_TIME 当前请求时间（时间戳）
START_MEM 开始运行时候的内存占用
REQUEST_METHOD 当前请求类型
IS_GET 当前是否GET请求
IS_POST 当前是否POST请求
IS_PUT 当前是否PUT请求
IS_DELETE 当前是否DELETE请求
IS_AJAX 当前是否AJAX请求
ENV_PREFIX 环境变量配置前缀
TMPL_PATH 用于改变全局视图目录
CLASS_APPEND_SUFFIX 是否追加类名后缀
```

助手函数.note

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
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

助手函数	描述
dump	浏览器友好的变量输出
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
route	注册路由

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],
```