ThinkPHP®

ThinkPHP5 开发手册

介绍

> 本手册内容已经过时，也不再更新
>
> 请阅读最新的 5.0完全开发手册 或者 5.0快速入门。

推荐阅读

> 《ThinkPHP5.0快速入门》是官方出品的学习和掌握 ThinkPHP5.0 不可多得的入门指引教程，针对新手用户由浅入深给出了详尽的使用。
>
> 本系列围绕WEB开发和API开发常用的一系列基础功能进行循序渐进的讲解。

快速入门

本章内容提供了ThinkPHP5.0的一些基本用法，并且处于不断完善过程。

> 注意：本章内容会随着最新版本的功能变化而做出一定的调整，因此相关功能的说明均针对Github的最新版本，而非官网下载版本。

GITHUB地址：https://github.com/top-think/think

安装配置

> 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

由于ThinkPHP5.0还在测试阶段，所以需要通过Git服务器下载，Git服务地址：https://github.com/top-think/think

> 下载或者使用GIT克隆到本地后，请（解压缩后）放置于你的WEB根目录下面的 tp5 子目录。

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：

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，那么请参考下面的列表检查下你的服务器环境：

- PHP5.4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

目录结构

下载最新版框架后，解压缩到web目录下面，可以看到初始的目录结构如下：

```
project  应用部署目录
├─composer.json         composer定义文件
├─README.md             README文件
├─build.php             自动生成定义文件（参考）
├─LICENSE.txt           授权说明文件
├─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）
│  ├─module             模块目录
│  │  ├─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
│  ├─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
├─public                WEB部署目录（对外访问目录）
│  ├─index.php          应用入口文件
│  ├─.htaccess          用于apache的重写
│  └─router.php         快速测试文件（用于自带webserver）
├─thinkphp              框架系统目录
│  ├─library            框架类库目录
│  │  ├─behavior        行为类库目录
│  │  ├─think           Think类库包目录
│  │  ├─org             Org类库包目录
│  │  ├─traits          系统Traits目录
│  │  ├─ ...            更多类库目录
│  ├─extend             扩展类库目录（可自定义）
│  ├─vendor             第三方类库目录
│  ├─mode               应用模式目录
│  ├─tpl                系统模板目录
│  ├─base.php           基础文件
│  ├─convention.php     框架惯例配置文件
│  └─start.php          框架引导文件
```

> router.php用于php自带webserver支持，可用于快速测试
> 进入public目录后，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，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规范参考。

系统架构

URL设计

ThinkPHP5.0在没有启用路由的情况下典型的URL访问规则是：

> http://serverName/应用（或应用入口文件）/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(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PATHINFO模式的访问地址，其中PATHINFO的分隔符是可以设置的。

注意：5.0取消了URL模式的概念，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/应用入口文件?s=/模块/控制器/操作/[参数名/参数值...]

首先，解释下其中的几个概念：

应用	基于同一个入口文件访问的项目我们称之为一个应用。（但应用可能具有多个入口）
模块	一个应用下面可以包含多个模块，每个模块在应用目录下面都是一个独立的子目录（小写）。
控制器	每个模块可以包含多个控制器，一个控制器通常体现为一个（控制器）类（驼峰法命名）。
操作	每个控制器类可以包含多个操作方法，每个操作是URL访问的最小单元。

简化URL访问

在ThinkPHP5.0中，出于优化的URL访问原则，我们还做出了如下的URL访问设计，这些设计包括：

隐藏应用入口文件 应用入口文件通常就是指index.php，可以通过URL重写隐藏。
隐藏应用入口文件index.php,以Apache为例说明如何设置。
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、把下面的内容保存为 .htaccess 文件放到应用入口文件的同级目录下

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

隐藏模块 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');
```

设置后，我们的URL访问地址则变成：

> http://serverName/应用入口/控制器/操作/[参数名/参数值...] // 访问的模块是index模块

隐藏控制器 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，只有一个模块，那么可以尝试使用单一模块结构，方法如下：
首先在入口文件中设置

```
// 关闭多模块设计
define('APP_MULTI_MODULE',false);
```

应用的目录结构就变成：

```
├─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/应用入口/控制器/操作/[参数名/参数值...]

单一模块设计的应用类库的命名空间有所调整，例如：

```
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。

命名规范

命名规范

ThinkPHP5的命名规范如下：

目录和文件

- 框架核心类库的目录统一使用小写规范，但应用目录名不强制规范，驼峰法和小写+下划线均支持，看团队规范；
- 类库、函数文件统一以`.php`为后缀；
- 类的文件名均以命名空间定义，并且命名空间的路径和类库文件所在路径一致（包括大小写）；
- 类名和类文件名保持一致，并统一采用驼峰法命名（首字母大写）

函数和类、属性命名

- 类的命名采用驼峰法，并且首字母大写，例如`User`、`UserType`，不需要添加controller、model等后缀，UserController直接更改为User；
- 函数的命名使用小写字母和下划线（小写字母开头）的方式，例如`get_client_ip`；
- 方法的命名使用驼峰法，并且首字母小写或者使用下划线“_”，例如`getUserName`，

`_parseType`，通常下划线开头的方法属于私有方法；

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

常量和配置

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

- 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`。

自动生成

自动生成

ThinkPHP5.0.0 具备自动创建功能，可以用来自动生成需要的模块和目录结构。

首先要在入口文件开启自动创建，如下：

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

开启后，需要定义自动创建的定义文件`build.php`。

默认的框架的根目录下面自带了一个`build.php`示例参考文件，把这个文件复制到application目录下面然后根据需要修改后就可以用于自动生成。

> 自动生成机制需要 application 目录具备可写权限。

默认的build.php内容如下：

```
return [
    // 生成运行时目录
    'runtime' => [
        '__dir__' => ['cache', 'log', 'temp','template'],
    ],
    // 定义index模块的自动生成
    'index'    => [
        '__file__'    => ['tags.php', 'user.php', 'hello.php'],
        '__dir__'     => ['behavior', 'controller', 'model', 'view'],
        'controller' => ['Index', 'Test', 'UserType'],
        'model'       => [],
        'view'        => ['index/index'],
    ],
    // 。。。 其他更多的模块定义
];
```

可以给每个模块定义需要自动生成的文件和目录，以及MVC类。

- `__dir__` 表示生成目录（支持多级目录）
- `__file__` 表示生成文件（不定义默认会生成 config.php 文件）
- controller 表示生成controller类
- model表示生成model类
- view表示生成html文件（支持子目录）

自动生成以 APP_PATH 为起始目录，`__dir__` 和 `__file__` 表示需要自动创建目录和文件，其他的则表示为模块自动生成。

模块的自动生成则以 `APP_PATH.'模块名/'` 为起始目录。

并且会自动生成模块的默认的Index访问控制器文件用于显示框架的欢迎页面。

我们还可以在 APP_PATH 目录下面自动生成其它的文件和目录，或者增加多个模块的自动生成，例如：

```
return [
    '__dir__'    => ['runtime/cache','runtime/log','runtime/temp'],
    '__file__'   => ['hello.php','test.php'],
    // 定义index模块的自动生成
    'index'    => [
        '__file__'    => ['tags.php', 'user.php', 'hello.php'],
        '__dir__'     => ['behavior', 'controller', 'model', 'view'],
        'controller' => ['Index', 'Test', 'UserType'],
        'model'       => [],
        'view'        => ['index/index'],
    ],
    // 定义test模块的自动生成
    'test'=>[
        '__dir__'    =>  ['behavior','controller','model','widget'],
        'controller'=>  ['Index','Test','UserType'],
        'model'     =>   ['User','UserType'],
        'view'      =>  ['index/index','index/test'],
    ],
    // 。。。 其他更多的模块定义
];
```

> 为了性能考虑，在生成完成之后，建议删除或者更名 `build.php` 文件。
> 将来计划添加扫描当前数据库自动生成模型类的功能。

引导文件

入口文件

入口文件是指应用的访问入口文件，由于ThinkPHP采用单一入口模式，支持多模块设计，因此多个模块的访问入口也是同一个。
入口文件唯一需要指定的就是应用的路径，然后加载框架的入口（引导）文件。例如：

```
// 定义项目路径
define('APP_PATH','../app/');
// 加载框架引导文件
require '../thinkphp/start.php';
// 执行应用
\\think\App::run();
```

ThinkPHP5用引导文件替代了旧版的框架入口文件，引导文件是可以根据环境和项目需要定制和调整的，而且引导文件的位置是可以随意放置。

引导文件

由于ThinkPHP5.0.0采用的是可分离式设计，因此，类库文件之间彼此相互独立，不是特别依赖，所以在项目开发的过程中，灵活和自由度较高，因此需要通过一定的组装和设置来完成，为了简化开发者进行实际的开发工作，引导文件就是起到这样的作用，预先定义和配置了一些规则。

引导文件一般来说，同时可以作为项目入口文件中框架的入口文件，例如，我们的项目入口文件定义如下：

```
define('APP_PATH',dirname(__DIR__).'/application/');
require dirname(__DIR__).'/thinkphp/start.php';
// 执行应用
\\think\App::run();
```

start.php就是系统自带的一个引导文件，包含了相关初始化和应用执行。

自动加载

概述

ThinkPHP5.0.0 真正实现了按需加载，所有类库采用自动加载机制，采用了两种方式来实现：命名空间和类库映射，并且支持composer类库的自动加载。

自动加载的实现由 `\think\Loader` 类库完成。

命名空间

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。
例如，如果我们实例化 `\think\log\driver\File` 类的话：

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

系统会自动加载 `thinkphp\library\think\log\driver\File.php` 文件。

如果实例化：

```
$class = new \org\UploadFile();
```

系统会自动加载 `thinkphp\library\org\UploadFile.php` 文件。

如果不清楚什么是命名空间，可以参考PHP手册相关部分。这里就不对命名空间的用法做过多的描述了。

系统对根命名空间的检测顺序如下：

1、优先检测是否存在注册过的根命名空间
2、检测composer自动加载的类库
3、然后检测核心目录（LIB_PATH）下是否存在根命名空间的对应子目录
4、检测是否应用类库(APP_PATH)命名空间
5、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

如果你需要额外的根命名空间的自动加载支持，可以首先注册根命名空间，例如：

```
\think\Loader::addNamespace('org',MY_PATH.'org/');
\think\Loader::addNamespace('com',MY_PATH.'com/');
```

> 注册的根命名空间优先，并且ThinkPHP5注册的命名空间根必须是小写。

注册新的命名空间支持后，我们就可以自动加载该命名空间下面的类库了。

类库映射

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

```
\think\Loader::addMap('think\Log',LIB_PATH.'think\Log.php');
\think\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：

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

> 通过类库映射的方式注册的类可以不遵循命名空间必须对应子目录的规范。

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：

```
\think\Loader::import('org.util.array');
\think\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
org	扩展类库包
com	企业类库包
@	表示当前模块类库包
vendor	第三方类库
traits	系统Traits类库

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。

配置

概述

配置的加载、设置和获取功能统一由\think\Config类完成。

使用

加载配置文件

```
\think\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：

```
\think\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：

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

```
\think\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：

```
$config = 'var1=val
var2=val';
\think\Config::parse($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。

设置配置参数

使用set方法动态设置参数，例如：

```
\think\Config::set('配置参数','配置值');
// 或者使用快捷方法
C('配置参数','配置值');
```

也可以批量设置，例如：

```
\think\Config::set(['配置参数1'=>'配置值','配置参数2'=>'配置值']);
// 或者使用
C(['配置参数1'=>'配置值','配置参数2'=>'配置值']);
```

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：

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

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

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,
],
```

如果需要使用独立配置文件的话，则首先在config.php中添加配置：

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

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

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

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

> 系统默认设置了两个独立配置文件，包括 database 和 route 。

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 \think\Config::get('配置参数1');
```

系统为get方法定义了一个快捷函数C，以上可以简化为：

```
echo C('配置参数1');
```

读取所有的配置参数：

```
dump(\think\Config::get());
 // 或者 dump(C());
```

或者你需要判断是否存在某个设置参数：

```
\think\Config::has('配置参数2');
```

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。

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

路由

概述

由于ThinkPHP5.0默认采用的URL规则是：

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

路由解析的最终结果通常是把URL地址解析到模块的某个控制器下的操作方法，在特殊的情况下，也可以跳转到外部地址或者执行闭包函数。

新版的路由功能做了大量的增强，包括：

- 支持路由到模块（模块/控制器/操作）、控制器（控制器类/操作）、类（任何类库）；
- 闭包路由的增强；
- 规则路由支持全局和局部变量规则定义（正则）；
- 支持路由到任意层次的控制器；
- 子域名路由功能改进；

- 路由分组并支持分组参数定义；
- 通过函数自定义路由检测规则；

ThinkPHP5.0的路由比较灵活，系统支持三种方式的URL解析规则：

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,
```

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，否则将抛出异常。

首页的路由规则是 `/` 。

注册路由规则

路由功能由 `think\Route` 类实现，包括路由注册和检测。

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

动态注册

使用Route类的register方法注册路由规则（通常可以在应用的公共文件中注册，或者定义配置文件后在公共文件中批量导入的方式注册），例如注册如下路由规则后：

```
\think\Route::register('new/:id','index/New/read');
```

我们访问：

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

> ThinkPHP5.0的路由规则定义是从根目录开始，而不是基于模块名的。

其实是访问的：

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

可以在register方法中指定请求类型，不指定的话默认为任何请求类型，例如：

```
\think\Route::register('new/:id','New/update','POST');
```

表示定义的路由规则在POST请求下才有效。
系统提供了为不同的请求类型定义路由规则的简化方法，例如：

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

如果要定义get和post请求支持的路由规则，也可以用：

```
\think\Route::register('new/:id','New/read','GET|POST');
```

我们也可以批量注册路由规则，例如：

```
\think\Route::register(['new/:id'=>'New/read','blog/:name'=>'Blog/detail']);
\think\Route::get(['new/:id'=>'New/read','blog/:name'=>'Blog/detail']);
\think\Route::post(['new/:id'=>'New/update','blog/:name'=>'Blog/detail']);
```

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

定义路由配置文件

如果不希望这么麻烦注册路由规则，可以直接在应用目录下面的 route.php 直接定义路由规则，内容示例如下：

```
return [

    '__pattern__' => [
        'name' => '\w+',
    ],
    'new/:id'       => 'New/read',
    '[blog]'     => [
        ':id'   => ['Blog/read', ['method' => 'get'], ['id' => '\d+']],
        ':name' => ['Blog/read', ['method' => 'post']],
    ],

];
```

__pattern__ 表示定义路由变量的全局规则。更详细的使用我们会在后面描述。

路由规则定义

路由定义由三个部分组成：路由表达式、路由地址和参数、路由响应的请求类型。

以register方法为例的话就是：

Think\Route::register('路由表达式','路由地址和参数','请求类型（默认为GET)','匹配参数（数组）','变量规则');

批量注册的时候规则是：

```
\think\Route::register(['路由表达式'=>'路由地址和参数',...],'','请求类型（默认为GET)','匹配参数（数组）','变量规则');
```

或者

```
\think\Route::get(['路由表达式'=>'路由地址和参数',...],'','匹配参数（数组）','变量规则'); // GET响应路由
\think\Route::post(['路由表达式'=>'路由地址和参数',...],'','匹配参数（数组）','变量规则'); // POST响应路由
\think\Route::put(['路由表达式'=>'路由地址和参数',...],'','匹配参数（数组）','变量规则'); // PUT响应路由
```

为了便于描述路由表达式和路由地址的对应关系，下面关于路由规则的描述定义均以批量定义的形式来表达。

路由表达式

路由表达式统一使用规则路由表达式定义，只能使用字符串。

> 正则路由定义功能已经废除，改由变量规则定义完成。

规则表达式

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

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

规则表达式的定义以“/”为参数分割符（无论你的URL_PATHINFO_DEPR设置是什么，请确保在定义规则表达式的时候统一使用“/”进行URL参数分割）。

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

完全匹配

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

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

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

会匹配成功,而

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

则不会匹配成功
如果是采用

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

方式定义的话，则两种方式的URL访问都可以匹配成功。

路由地址及参数

路由地址和额外参数表示前面的路由表达式最终需要路由到的地址以及一些需要的参数，支持下面4种方式定义：

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

路由到模块/控制器/操作

这是最常用的一种路由地址定义，路由地址中的 [模块/控制器/操作] 的解析规则是先从后面的操作开始解析，然后依次往前解析控制器和模块，如果没有则用默认控制器及默认模块。

> 如果关闭路由功能，那么解析规则其实就是采用该方式。

路由地址中支持多级控制器，使用下面的方式进行设置：

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

表示路由到下面的控制器类，

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

支持可以动态改变路由地址，例如：

```
'blog/:action'=>'index/blog/:action'
```

路由到重定向地址

重定向的外部地址必须以“/”或者http开头的地址。

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

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

和

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

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

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

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

路由地址采用重定向地址的话，如果要引用动态变量，直接使用动态变量即可。

采用重定向到外部地址通常对网站改版后的URL迁移过程非常有用，例如：

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

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

路由到控制器的方法

这种方式看起来似乎和第一种是一样的，本质的区别是直接执行某个控制器类的方法，而不需要去解析 模块/控制器/操作 这些。
例如，定义如下路由后：

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

系统会直接执行

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

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

路由到类的方法

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

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

执行的是 \app\index\service\Blog类的read方法。
也支持执行某个静态方法，例如：

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

额外参数

在路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐式传入需要的操作中，

有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

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

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

匹配参数

匹配参数用于设置当前的路由规则详细的匹配条件，匹配参数是优先于路由表达式检测的，目前支持设置的匹配参数包括：

参数名	参数说明
method	请求类型
ext	伪静态后缀
domain	域名检测
https	是否https访问
callback	自定义检测，可以支持所有is_callable的定义，包括闭包
before_behavior	前置行为（检测）
after_behavior	后置行为（执行）

例如我们可以设置当前的路由规则只有当伪静态后缀为html的时候才会匹配：

```
\think\Route::register('new/:id','New/read','GET',['ext'=>'html']);
```

或者

```
\think\Route::get('new/:id','New/read',['ext'=>'html']);
```

设置后，URL访问地址：

```
http://serverName/new/6
```

将无法正确匹配，所以会报错。只有访问：

```
http://serverName/new/6.html
```

的时候才能正常访问。

可以支持设置多个后缀，例如：

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

如果设置如下：

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

那么访问

```
https://serverName/new/6
```

才是有效匹配的。
如果定义了

```
\think\Route::get('new/:id','New/read',['callback'=>'checkRoute']);
```

则调用自定义的checkRoute方法进行检测是否匹配当前路由。

变量规则

ThinkPHP5.0支持在规则路由中为变量用正则的方式指定变量规则，弥补了之前变量规则太局限的问题，并且支持全局规则设置。使用方式如下：

设置全局变量规则，全局路由有效：

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

局部变量规则，仅在当前路由有效：

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

如果一个变量同时定义了全局规则和局部规则，局部规则会覆盖全局变量的定义。

完整URL规则

如果要对整个URL进行变量规则检查，可以进行 `__url__` 变量规则（效果相当于之前版本的使用正则路由规则检测），例如：

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

URL映射

URL映射其实属于URL路由的静态简化版，由于不进行路由规则的遍历操作而是直接定位，因此效率较高，但也因为是类似于静态路由的概念，从而作用有限。
注册URL映射的方法如下：

```
\think\Route::map('new/top','news/index?type=top');
```

定义之后，如果我们访问：

```
http://serverName/new/top
```

其实是访问：

```
http://serverName/index/news/index/type/top
```

和路由匹配不同的是，URL映射匹配是完整匹配，所以如果访问：

```
http://serverName/new/top/var/test
```

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

URL映射定义也支持批量方式：

```
\think\Route::map(['new/top'=>'news/index?type=top','blog/new'=>'Blog/index?type=new']);
```

因为URL映射中的映射规则是静态定义，所以不能含有动态变量，而且在匹配的时候必须是完全匹配，所以下面的定义是不支持的或者无法生效的：

```
\think\Route::map('new/:id','news/read');
```

URL映射无法支持匹配请求类型。

> 如果你绑定了模块那么，URL映射会自动加上模块名。

闭包定义支持

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

```
\think\Route::get('test',function(){ echo 'just test';});
\think\Route::get('hello/:name',function($name){ echo 'Hello,'.$name;});
```

参数传递

闭包定义的时候支持参数传递，例如：

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

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

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

则浏览器输出的结果是：

```
Hello,thinkphp
```

资源路由

5.0支持设置RESTFul请求的资源路由，方式如下：

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

或者在路由配置文件中使用 `__rest__` 添加资源路由定义：

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

设置后会自动注册7个路由规则如下

请求类型	生成路由规则	对应操作方法
GET	`blog`	index
GET	`blog/create`	create
POST	`blog`	save

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

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

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

方法如下：

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

    public function read($id){
    }

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

可以改变默认的id参数名，例如：

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

控制器的方法定义调整如下：

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

    public function read($blog_id){
    }

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

也可以在定义资源路由的时候限定执行的方法，例如：

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

如果需要更改某个资源路由的对应操作，可以使用下面方法：

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

设置之后，URL访问变为：

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

创建blog页面的对应的操作方法也变成了add。

支持批量更改，如下：

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

> 注意，rest数据定义的索引 `save/update/delete` 是不能更改的。

资源嵌套

支持资源路由的嵌套，例如：

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

就可以访问如下地址：

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

生成的路由规则分别是：

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

Comment控制器对应的操作方法如下：

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

如果需要改变其中的变量名，可以使用：

```
\think\Route::resource('blog.comment','index/comment',['var'=>['blog'=>'blogId']]);
```

Comment控制器对应的操作方法改变为：

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

路由分组

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

例如，我们有定义如下两个路由规则的话

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

可以合并到一个blog分组

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

可以使用Route类的group方法进行注册，如下：

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

可以给分组路由定义一些公用的路由设置参数，例如：

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

URL生成

可以统一使用 \think\Url类生成URL地址，例如：

```
\think\Url::build('hello');
\think\Url::build('index/hello');
// 或者使用帮助函数
U('hello');
U('index/hello');
```

> 如果传入的url地址存在路由定义，会自动转换为路由定义。

控制器

访问控制器

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

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

```
namespace app\index\controller;
class New
{
    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/new/index // 输出 index
http://serverName/index/new/add     // 输出 add
http://serverName/index/new/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。

渲染模板和输出

默认的情况下，如果不需要渲染模板，无需继承 `\think\Controller` 类，如果需要进行模板渲染等操作，可以改为：

```
namespace app\index\controller;
use think\Controller;
class New extends Controller
{
    public function index(){
        return $this->fetch();
    }
    public function add(){
        return $this->fetch();
    }
    public function edit($id){
        return $this->show('edit:'.$id);
    }
}
```

> 新版控制器一般不需要任何输出，直接return即可。

多层控制器

新版支持任意层次的控制器，并且支持路由，例如：

```
namespace app\index\controller\one;
use think\Controller;
class New 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.new/index
```

空操作和空控制器

空操作

空操作是指系统在找不到指定的操作方法的时候，会定位到空操作（ `_empty` ）方法来执行，利用这个机制，我们可以实现错误页面和一些URL的优化。
例如，下面我们用空操作功能来实现一个城市切换的功能。

我们只需要给City类定义一个 `_empty` （空操作）方法：

```
<?php
namespace app\index\controller;
class City {
    public function _empty($name){
        //把所有城市的操作解析到city方法
        return $this->showCity($name);
    }

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

接下来，我们就可以在浏览器里面输入

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

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

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

空控制器

空控制器的概念是指当系统找不到指定的控制器名称的时候，系统会尝试定位空控制器(Error)，利用这个机制我们可以用来定制错误页面和进行URL的优化。
现在我们把前面的需求进一步，把URL由原来的

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

变成

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

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

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

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

空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

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 New 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：

```
namespace app\index\controller;
 use think\controller\Rest;
 class New 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：

定义规范	**说明**
操作名*提交类型*资源后缀	标准的Restful方法定义，例如 read_get_pdf
操作名_资源后缀	当前提交类型和restDefaultMethod相同的时候，例如read_pdf
操作名_提交类型	当前资源后缀和restDefaultType相同的时候，例如read_post

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

```
namespace app\index\controller;
 use think\controller\Rest;
 class Info extends Rest{
   Public function read_get_xml($id){
   // 输出id为1的Info的XML数据
 }
   Public function read_xml($id){
   // 输出id为1的Info的XML数据
 }
   Public function read_json($id){
   // 输出id为1的Info的json数据
 }
 }
```

如果我们访问的URL是：

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

假设我们没有定义路由，这样访问的是Info模块的read操作，那么上面的请求会调用Info类的read_get_xml方法，而不是read_xml方法，但是如果访问的URL是：

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

那么则会调用read_json方法。

分层控制器

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

例如，我们定义New事件控制器如下：

```
namespace app\index\event;
class New {
    public function insert(){
        echo 'insert';
    }
    public function update($id){
        echo 'update:'.$id;
    }
    public function delete($id){
        echo 'delete:'.$id;
    }
}
```

定义完成后，就可以用下面的方式实例化并调用方法了：

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

为了方便调用，系统提供了A快捷方法直接实例化多层控制器，例如：

```
$Event = A('New','event');
$Event->update(5); // 输出 update:5
$Event->delete(5); // 输出 delete:5
```

支持跨模块调用，例如：

```
$Event = A('Admin/New','event');
$Event->update(5); // 输出 update:5
```

表示实例化Admin模块的New控制器类，并执行update方法。

除了实例化分层控制器外，还可以直接调用分层控制器类的某个方法，例如：

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

也可以使用快捷R方法实现相同的功能：

```
$result = R('New/update',['id'=>5],'event'); // 输出 update:5
```

利用分层控制器的机制，我们可以用来实现Widget（其实就是在模板中调用分层控制器），例如：
定义 `index\widget\New` 控制器类如下：

```
namespace \app\index\widget;
class New {
    public function header(){
        echo 'header';
    }
    public function left(){
        echo 'left';
    }
    public function menu($name){
        echo 'menu:'.$name;
    }
}
```

我们在模板文件中就可以直接调用 `app\index\widget\New` 分层控制器了，例如：

```
<?php R('New/header','','widget');?>
<?php R('New/menu',['name'=>'think'],'widget');?>
```

框架还提供了W方法用于简化Widget控制器的调用，例如可以直接使用：

```
<?php W('New/header');?>
<?php W('New/menu',['name'=>'think']);?>
```

模型

概述

模型类Think\Model配合数据库中间层Think\Db实现了完整的ORM功能，包括CURD和ActiveRecord实现。
基础模型类Model的设计非常灵活，无需进行任何模型定义，就可以进行相关数据表的ORM和CURD操作，只有在需要封装单独的业务逻辑的时候，模型类才是必须被定义的。
新版采用了PHP的Trait特性实现了模型的动态组装，可以更加灵活的实现模型的扩展。

模型定义

如果你仅仅需要实现对数据表的CURD操作的话，实际上根本不需要定义模型类，直接实例化基础模型类即可。（参考模型实例化）

只有当你需要额外定义模型的属性或者方法逻辑的时候，才需要额外定义模型类。模型类一般位于模块的 model 目录下面，例如：

```
namespace app\index\model;
use think\Model;
class New extends Model{
    public function getNews(){
        //添加自己的业务逻辑
         // ...
    }
}
```

实例化代码如下：

```
$model = \think\Loader::model('index/New');
// 快捷方法
$model = D('index/New');
```

如果你的模型没有定义模型类的话，可以直接使用

```
$model = \think\Loader::table('New');
// 快捷方法
$model = M('New');
```

并且支持传入模型参数：

```
$config = [
    'prefix'    => 'think_',
    'connection'=>  $connection,
    ...
];
// 或者使用
$model = \think\Loader::table('New',$config);
```

支持的配置参数包括：

参数名	含义
prefix	数据表前缀
connection	数据库连接信息
table_name	实际的数据表（不含前缀）
true_table_name	实际的数据表（含前缀 支持指定数据库名）
db_name	数据库名称

支持多层的模型类定义，例如：

```
namespace app\index\model\one;
use think\Model;
class New extends Model{
    public function getNews(){
        //添加自己的业务逻辑
         // ...
    }
}
```

实例化代码如下：

```
$model = \think\Loader::model('index/one/New');
// 快捷方法
$model = D('index/one/New');
```

注意如果类名是驼峰方式的话，例如：

```
namespace app\index\model;
use think\Model;
class UserType extends Model{
}
```

对应的模型文件名应该是：

```
application\index\model\UserType.php
```

默认情况下，模型类和数据表的默认对应关系如下：

模型名（类名）	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。
在ThinkPHP的模型里面，有几个关于数据表名称的属性定义：

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

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

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

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

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

注意trueTableName需要完整的表名定义

除了数据表的定义外，还可以对数据库进行定义，例如：

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

查询语言

ThinkPHP5.0的查询语言基本和3.2版本保持一致，核心\Think\Model除了基本的CURD和AR查询之外，还提供了一些统计函数、getField方法，及动态查询方法，使用如下：

```
$User = D('User');
$User->count();
$User->getField('name');
$User->getByName('thinkphp');
$User->getFieldByName('thinkphp','name');
```

> 查询语言基本和3.2版本没有任何变化，请先参考3.2的完全开发手册。

自动验证和自动完成

要使用模型的自动验证和自动完成功能，需要引入traits，例如：

```
namespace app\index\model;
T('model/Auto');
class User extends \think\Model{
    use \traits\model\Auto;

    // 下面是模型类的方法和属性定义
}
```

如果你的PHP版本是5.5的话，可以省略下面这行代码：

```
T('model/Auto');
```

视图模型

视图模型的用法也是需要首先继承视图模型扩展，例如：

```
namespace app\index\model;
use think\model\View;
class UserType extends View{
}
```

其它用法和之前的视图模型用法一致。

关联模型

```
namespace app\index\model;
use think\model\Relation;
class User extends Relation{
}
```

视图

视图功能由Think\View类和模板引擎（驱动）类一起完成。

实例化视图类

```
// 实例化视图类
$view = new \think\View();
 // 渲染模板输出
return $view->fetch();
```

如果你的控制器继承了\think\Controller类的话，则可以直接使用

```
 // 渲染模板输出
return $this->fetch();
```

> 需要注意的是，ThinkPHP5的视图fetch方法不会直接渲染输出，只是返回解析后的内容。如果在控制器类返回 视图解析内容的话，渲染输出系统会自动调用think\Response类的send方法进行渲染输出。

模板定位规则

模板文件目录默认位于模块的view目录下面，视图类的fetch方法中的模板文件的定位规则如下：

如果调用没有任何参数的fetch方法：

```
return $view->fetch();
```

则按照系统的默认规则定位模板文件到：

> [模板文件目录]/当前控制器名（小写）/当前操作名（小写）.html

如果（指定操作）调用：

```
return $view->fetch('add');
```

则定位模板文件为：

> [模板文件目录]/当前控制器名/add.html

如果调用控制器的某个模板文件使用：

```
return $view->fetch('user/add');
```

则定位模板文件为：

[模板文件目录]/user/add.html

全路径模板调用：

```
return $view->fetch(MODULE_PATH.'view/public/header.html');
```

则定位模板文件为：

MODULE_PATH.'view/public/header.html'

模板主题支持

默认并不支持模板主题功能，需要配置开启，例如：

```
$view = new \think\View();
return $view->theme('blue')->fetch('User/add');
```

表示使用blue模板主题风格输出，定位模板文件为：

[模板文件目录]/blue/User/add.html

或者也可以使用下面的方式支持模板主题：

```
$view = new \think\View();
return $view->fetch('blue/User/add');
```

和上面使用模板主题功能输出的模板文件是相同的。

如果要进行自动侦测模板主题的话，按照如下方式使用：

```
$view = new \think\View();
return $view->theme(true)->fetch('User/add');
```

theme(true) 表示开启模板主题自动侦测功能，通常是在URL地址中传入模板主题参数，默认情况下，只需要使用：

```
http://serverName/moduleName/controllerName/actionName/t/blue
```

之后，模板主题名称将会被保存到Cookie中，直到重新指定新的模板主题名称。

设置输出参数

可以在视图实例化的时候设置参数或者在模板输出之前动态设置参数，例如下面两种方式等效：

```
// 实例化视图类的时候设置参数
$view = new \think\View(['view_suffix'=>'.html','view_depr'=>'/']);
 // 或者使用http方法动态设置视图输出参数
$view->config(['view_suffix'=>'.html','view_depr'=>'_']);
```

支持独立设置某个参数，例如：

```
$view->config('view_suffix','.html');
```

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->config('parse_str',['__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：

```
'parse_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。

设置模板引擎

使用View类的engine方法设置当前视图输出的模板解析引擎，如果没有指定模板引擎的话 默认为PHP自身模板。

```
// 设置模板引擎
return $view->engine('think',['tpl_path'=>MODULE_PATH.'view/','cache_path'=>MODULE_PATH.'cach
e/','compile_type'=>'File'])->fetch();
```

模板引擎的参数根据不同的模板引擎而异。

视图类的模板引擎驱动位于 think\view\driver\ 目录下面，可以自行扩展。
Think模板引擎的驱动类实现如下：

```
namespace think\view\driver;
 use think\Template;
 class Think {
    private $template   =   null;
    public function __construct($config=[]){
        $this->template =   new Template($config);
    }
    public function fetch($template,$data=[],$cache=[]){
        if(is_file($template)) {
            $this->template->display($template,$data,$cache);
        }else{
            $this->template->fetch($template,$data);
        }
    }
 }
```

模板引擎驱动类需要实现的接口方法就是fetch方法。

如果你不需要任何模板引擎，直接使用原生PHP作为模板解析的话，可以使用

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

如果全局使用的话，还可以直接配置：

```
'engine_type'       => 'php',
```

视图绑定数据

绑定数据到模板输出有三种方式：

1、使用assign方法给模板变量赋值：

```
// 实例化视图类
$view = new \think\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();
```

2、直接对象赋值

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

3、在输出模板的时候传入模板变量

例如：

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

其他设置

如果希望直接解析内容而不通过模板文件的话，可以使用：

```
return $view->show($content,$vars);
```

指定当前的视图模板文件目录：

```
return $view->config('view_path',MODULE_PATH.'tpl/')->fetch();
```

模板

5.0对模板引擎进行了重构，主要改进如下：

兼容原来所有的标签功能和用法，已 对正则进行了优化，标签库和内置的普通标签可以使用一样的边界符，比如都用"{}"，只要不重名不会相互干扰，这样这些标签就可以和html标签区分开。（默认标签库和变量标签配置都采用统一的定界符{和}）

模板支持多级继承

C继承B，而B又继承了A，C中的block会覆盖B和A中的同名block。

include标签支持多层嵌套，可以传变量。

如：

```
{include file="Public/nav" selected="{$id}" }
```

在Public/nav模板用[selected]得到的是[$id}被解析后的值，而在3.2版中这样的写法是不能正确得到{$id}的值的。

增强了.语法的应用范围

`{$user.name.$group.name}` 解析后是

```
<?php echo $user['name'].$group['name']; ?>
```

`{:substr($varname.aa, $varname.bb)}` 解析后是

```
<?php echo substr($varname['aa'], $varname['bb']); ?>
```

`.` 语法在各个标签中都可以使用，$a.b.c这样的形式都能正确解析成$a['b']['c']

自定义标签库

自定义TagLib标签库

兼容原有(3.2版本)自定义标签功能

标签库定义

```
<?php
namespace tools\taglib;

use think\template\TagLib;

/**
 *
 */
class Mytag extends TagLib
{
    protected $tags = [
        'test' => ['attr' => 'name,value', 'close' => 0]
        // attr : 自定义标签的属性, close : 是否闭合标签,下面有说明
    ];

    public function _test($tags, $content)
    {
        $name  = $tags['name'];
        $value = $tags['value'];
        // 逻辑代码
        return 'something';
    }
}
```

标签库使用

标签库默认情况下不会自动加载,如需要自动加载,需要在配置文件中设置

```
    'template'              => [
        'taglib_pre_load' => 'tools\\taglib\\Sys'
    ]
```

这样在模板中就可以调用

1.标签定义close为0的时候

```
{Mytag:test name="" value="" /}
```

2.标签定义close为1的时候

```
{Mytag:test name="" value=""}something{/Mytag:test}
```

数据库

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

> 操作，而无需针对不同的数据库写不同的代码和底层实现，Db类会自动调用相应的数据库驱动来处理。采用PDO方式，目前包含了Mysql、SqlServer、PgSQL、Sqlite、Oracle、Mongo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、全局配置定义

常用的配置方式是在公共配置文件或者模块配置文件中添加下面的配置参数：

```
'database'=> [
    // 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'       => true,
    // 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'    => '',
];
```

也支持独立数据库配置文件定义的方式，例如在 application/database.php文件中定义如下：

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

也可以支持简化的DSN数据库连接字符串写法（支持大部分的连接参数），例如：

```
'database'=> 'mysql://root:passwd@127.0.0.1:3306/thinkphp#utf8'
```

如果需要的话，可以对每个模块定义不同的数据库连接信息，只需要在模块配置文件中添加database设置参数即可。

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，例如：

如果需要使用长连接，可以采用下面的方式定义：

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

> 你可以在params里面配置任何PDO支持的连接参数。

二、模型类定义

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

```
//在模型里单独设置数据库连接信息
namespace app\index\model;
use Think\Model;
class User extends Model{
    protected $connection = [
        // 数据库类型
        'type'        => 'mysql',
        // 数据库连接DSN配置
        'dsn'         => '',
        // 服务器地址
        'hostname'    => '127.0.0.1',
        // 数据库名
        'database'    => 'thinkphp',
        // 数据库用户名
        'username'    => 'root',
        // 数据库密码
        'password'    => '',
        // 数据库连接端口
        'hostport'    => '',
        // 数据库连接参数
        'params'      => [],
        // 数据库编码默认采用utf8
        'charset'     => 'utf8',
        // 数据库表前缀
        'prefix'      => 'think_',
        // 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，定义格式为：

数据库类型://用户名:密码@数据库地址:数据库端口/数据库名#字符集

例如：

```
//在模型里单独设置数据库连接信息
namespace app\index\model;
use think\Model;
class User extends Model{
    //或者使用字符串定义
    protected $connection = 'mysql://root:1234@127.0.0.1:3306/thinkphp#utf8';
}
```

> 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';
```

那么，我们可以把模型类的属性定义改为：

```
//在模型里单独设置数据库连接信息
namespace app\index\model;
use think\Model;
class User extends Model{
    //调用配置文件中的数据库配置1
    protected $connection = 'db_config1';
}
```

```
//在模型里单独设置数据库连接信息
namespace app\index\model;
use think\Model;
class Info extends Model{
    //调用配置文件中的数据库配置1
    protected $connection = 'db_config2';
}
```

三、实例化定义

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

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

表示实例化User模型，连接的是demo数据库的other_user表，采用的连接信息是第三个参数配置的。如果我们在项目配置文件中已经配置了 db_configG2 的话，也可以采用：

```
$User = \think\Loader::model('User','other_','db_config2');
```

> 需要注意的是，ThinkPHP的数据库连接是惰性的，所以并不是在实例化的时候就连接数据库，而是在有实际的数据操作的时候才会去连接数据库。

四、应用状态数据库配置

ThinkPHP5.0支持给不同的应用状态配置不同的数据库连接信息，例如，我们有一个应用状态（app_status）home 是用于家里的工作连接，可以配置如下：

```
// 开启数据库状态切换
'use_db_switch'         => true,
'database'              => [
    'default'=>[
        // 数据库类型
        'type'        => 'mysql',
        // 数据库连接DSN配置
        'dsn'         => '',
        // 服务器地址
        'hostname'    => '192.168.1.1',
        // 数据库名
        'database'    => 'office',
        // 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,
    ],
    'home'=>[
        // 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'    => 'home',
        // 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,
    ],
]
```

default配置用于 app_status 为空的情况下的数据库连接，home配置则是用于 app_status 切换到 home 的时候的自动连接。

域名部署

概述

在ThinkPHP5中，可以很轻松的实现将模块部署到某个域名规则（包括完整域名、二级域名、三级域名和泛域名），而不需要分开不同的应用入口文件。

要使用域名部署，首先需要在应用的公共配置文件中开启

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

域名部署

开启域名部署后，可以在应用配置文件中设置 url_domain_rules 或者通过下面的方法动态设置。

注册域名部署规则

可以用Route类的domain方法注册子域名部署规则：

```
\think\Route::domain('域名规则','部署规则');
```

或者批量注册：

```
\think\Route::domain(['域名规则'=>'部署规则',...]);
```

域名规则

域名规则的定义可以支持完整域名（或者IP）、二级域名（包括泛二级域名）和三级域名（包括泛三级域名），例如：

```
\think\Route::domain('doc.thinkphp.cn','home/doc'); // 部署完整域名doc.thinkphp.cn到应用的home模块的doc控制器
\think\Route::domain('202.65.34.5','admin'); // IP地址202.65.34.5访问部署到admin模块
\think\Route::domain('blog','home/blog'); // 部署blog二级域名到home模块的blog控制器
\think\Route::domain('*','home'); // 泛二级域名部署到home模块
\think\Route::domain('admin.blog','admin'); // 三级域名部署到admin模块
\think\Route::domain('*.user','user'); // 泛三级域名部署到user模块
```

或者采用批量注册方式定义：

```
\think\Route::domain([
    'doc.thinkphp.cn'   =>  'home/doc',
    '202.65.34.5'       =>  'admin',
    'blog'              =>  'home/blog',
    '*'                 =>  'home',
    'admin.blog'        =>  'admin',
    '*.user'            =>  'user'
 ]);
```

注意域名部署的优先判断级别，依次是：

完整域名（或IP）-> 二级域名（或者三级域名）-> 泛三级域名 -> 泛二级域名

部署规则

域名的部署规则其实就是把基于部署域名的URL访问绑定到相关地址，支持下面几种类型，包括：

- 绑定到模块/控制器/操作；
- 绑定到命名空间；
- 绑定到类；
- 绑定到路由分组；
- 绑定闭包方法；

> 需要注意的是，除了了模块绑定之外的类型，将不会再进行路由检测。

模块绑定

也就是把域名绑定到指定的模块、控制器，甚至是操作的方式，并且还可以支持传入额外的参数，例如：

```
\think\Route::domain('blog','home/blog?status=1&type=1'); // 部署blog二级域名到home/blog，并传入status和type参数
```

上面的定义会把 `$_GET['status'] = 1` 和 `$_GET['type'] = 1` 隐式传入当前的访问。
如果使用的是泛域名规则的话，可以获取泛域名的内容作为参数带入，例如：

```
\think\Route::domain('*.user','user?status=1&username=*'); // 泛三级域名部署到User模块
```

上面的定义，表示部署泛三级域名*.user到user模块，并传入 `$_GET['status'] = 1` 和 `$_GET['username'] = '当前实际的泛域名'`。

如果成功匹配到某个域名部署规则的话，URL地址中的解析规则将会发生变化，例如，采用域名部署之前的URL地址是：

```
http://www.domain.com/admin/blog/add
```

如果定义了

```
\think\Route::domain('admin','admin');
```

那么访问的URL地址就变成了：

```
http://admin.domain.com/blog/add
```

绑定到命名空间

可以支持绑定到命名空间，例如：

```
\think\Route::domain('admin','\app\admin\api');
```

把admin子域名的访问绑定到 `\app\admin\api` 命名空间下面，如果请求URL地址是：

```
http://admin.domain.com/index.php/user/info/id/8
```

那么执行的其实就是 \app\admin\api\user 类的info方法，并且传入参数id=8

绑定到类

支持直接绑定到类，例如：

```
\think\Route::domain('admin','@\app\admin\api\user');
```

把admin子域名的访问绑定到 `\app\admin\api\user` 类下面，如果请求URL地址是：

```
http://admin.domain.com/index.php/info/id/8
```

那么执行的其实就是和上面相同的方法。

绑定到路由分组

如果你定义了较多的分组路由，也可以支持把域名绑定到某个路由分组，例如：

```
\think\Route::domain('admin','[admin]');
```

绑定之后，检测路由的时候只会检测该分组下面的路由。

闭包支持

域名的部署规则支持闭包函数，例如：

```
\think\Route::domain('*',function(){
    exit('非法访问！');
});
```

部署规则中的闭包函数不支持外部传入参数，但可以支持默认参数。

如果要进行路由劫持，可以在闭包函数里面返回数组，例如：

```
\think\Route::domain('user',function(){
    // 表示路由到user模块的index控制器
    return ['type'=>'module','module'=>'user/index'];
});
```

配置文件

也可以直接在应用配置文件中直接设置如下：

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

在route.php文件中添加域名部署定义如下：

```
return [
    '__domain__'=>[
        'doc.thinkphp.cn'   =>  'home/doc',
        '202.65.34.5'       =>  'admin',
        'blog'              =>  'home/blog',
        '*'                 =>  'home',
        'admin.blog'        =>  'admin',
        '*.user'            =>  'user'
    ]
];
```

输入

概述

系统提供了 `\think\Input` 类来完成输入变量的获取和安全过滤。

变量获取

获取GET变量

```
\think\Input::get('id'); // 获取某个get变量
\think\Input::get('name'); // 获取get变量
\think\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：

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

获取POST变量

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

使用快捷方法实现：

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

获取PUT变量

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

使用快捷方法实现：

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

获取PARAM变量

PARAM变量是框架提供的用于自动识别GET、POST或者PUT请求的一种变量获取方式，例如：

```
\think\Input::param('name');
如果当前是get请求，那么等效于
\think\Input::get('name');
如果当前是post请求，则等同于
\think\Input::post('name');
```

使用快捷方法实现：

```
I('param.name');
I('param.');
或者
I('name');
I('');
```

因为I函数默认就采用PARAM变量读取方式。

获取REQUEST变量

```
\think\Input::request('id'); // 获取某个request变量
\think\Input::request(); // 获取全部的request变量
```

使用快捷方法实现：

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

获取SERVER变量

```
\think\Input::server('PHP_SELF'); // 获取某个server变量
\think\Input::server(); // 获取全部的server变量
```

使用快捷方法实现：

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

获取SESSION变量

```
\think\Input::session('user_id'); // 获取某个session变量
\think\Input::session(); // 获取全部的session变量
```

使用快捷方法实现：

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

获取Cookie变量

```
\think\Input::cookie('user_id'); // 获取某个cookie变量
\think\Input::cookie(); // 获取全部的cookie变量
```

使用快捷方法实现：

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

变量过滤

支持对获取的变量进行过滤，过滤方式包括函数、方法过滤，以及PHP内置的Types of filters，例如：

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

可以支持传入多个过滤规则，例如：

```
\think\Input::param('username','strip_tags,strtolower'); // 获取param变量 并依次调用strip_tags、strtolower函数过滤
```

Input类还支持PHP内置提供的Filter ID过滤，例如：

```
\think\Input::post('email',FILTER_VALIDATE_EMAIL);
```

框架对FilterID做了转换支持，因此也可以使用字符串的方式，例如：

```
\think\Input::post('email','email');
```

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

具体的字符串根据filter_list函数的返回值来定义。

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

例如，

```
\think\Input::post('email',FILTER_VALIDATE_EMAIL,'');
```

就表示，如果不是规范的email地址的话 返回空字符串。

变量修饰符

I函数支持对变量使用修饰符功能，可以更好的过滤变量。

用法如下：
I('变量类型.变量名/修饰符'); 或者
\think\Input::get('变量名/修饰符');

例如：

```
I('get.id/d');
I('post.name/s');
I('post.ids/a');
\think\Input::get('id/d');
```

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

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

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

缓存

概述

ThinkPHP5.0采用Think\Cache类统一对缓存功能提供支持。

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。

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

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。

设置缓存

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

删除缓存

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

清空缓存

```
\think\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：

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

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

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

日志

日志记录由 \think\Log 类完成。

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。

```
\think\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/' 。

> 不同的日志类型可能会使用不同的初始化参数。

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

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,
],
```

> File方式记录，对应的驱动文件位于系统的 library/think/log/driver/File.php 。

手动记录

一般情况下，系统的日志记录是自动的，无需手动记录，但是某些时候也需要手动记录日志信息，Log类提供了3个方法用于记录日志。

方法	描述
Log::record()	记录日志信息到内存
Log::save()	把保存在内存中的日志信息（用指定的记录方式）写入
Log::write()	实时写入一条日志信息

由于系统在请求结束后会自动调用Log::save方法，所以通常，你只需要调用Log::record记录日志信息即可。

record方法用法如下：

```
\think\Log::record('测试日志信息');
```

默认的话记录的日志级别是INFO，也可以指定日志级别：

```
\think\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：

```
\think\Log::write('测试日志信息，这是警告级别，并且实时写入','notice');
```

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。

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
\think\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。

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_begin	视图输出开始标签位
view_filter	视图输出过滤标签位
action_end	控制器结束标签位
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：

```
\think\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：

```
\think\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：

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

```
\think\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：

```
\think\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：

```
\think\Hook::listen('action_init',$params);
\think\Hook::add('action_init',function($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
$result = \think\Hook::exec('behavior\\CheckAuth','run',$params);
```

多语言

ThinkPHP内置多语言支持，如果你的应用涉及到国际化的支持，那么可以定义相关的语言包文件。任何字符串形式的输出，都可以定义语言常量。

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：

```
'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。

```
// 设定当前语言
\think\Lang::range('zh-cn');
// 或者进行自动检测语言
\think\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：

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

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

ThinkPHP语言文件定义采用返回数组方式：

```
return [
    'hello thinkphp'=>'欢迎使用ThinkPHP',
    'data type error'=>'数据类型错误',
];
```

也可以在程序里面动态设置语言定义的值，使用下面的方式：

```
\think\Lang::set('define2','语言定义');
$value = \think\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，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','maximum' => '2MB'])}
```

调试

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
\think\Log::record('调试信息','日志类型');
// 直接写入调试信息
\think\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'=> THINK_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

基本 文件 错误 SQL 调试

1. 请求信息 : 2015-12-21 18:22:59 HTTP/1.1 GET : localhost/tp5/public/index.php/
2. 运行时间 : 0.009313s [吞吐率 : 107.38req/s]
3. 内存消耗 : 552.38kb
4. 查询信息 : 0 queries 0 writes
5. 缓存信息 : 0 reads,0 writes
6. 文件加载 : 20
7. 配置加载 : 48
8. 会话信息 : SESSION_ID=4058vf1tt1uc6ffaslui2s3bs1

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

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。

新特性介绍

新版ThinkPHP核心基于PHP5.4版本开发，完全重构的核心功能和架构设计，并增加了众多的新特性，下面我们来一一分析下新版的这些比较重要的特性，包括：

- 接近完美的路由功能；
- 更灵活的控制器；
- API开发支持；
- 全新的文件自动生成；
- PHP的Traits特性的引入；
- Loader自动加载和实例化；
- 内置整合方便远程调试的SocketLog；

增强路由功能

新版的路由经过重新设计，功能大为增强，主要特性有：

动态注册路由

除了在路由配置文件中定义，现在可以通过\Think\Route类的静态方法注册路由。

最关键的方法是register方法，用法如下：

```
// 除了路由规则和路由地址外，其他都是可选
\Think\Route::register('路由规则','路由地址'[,'请求类型','路由参数','变量规则']);
```

如果没有传入请求类型的话，默认的是任何请求类型都有效，可以指定其他请求类型：

```
\Think\Route::register('new/:id','New/update','POST');
```

如果要定义get和post请求支持的路由规则，也可以用：

```
\Think\Route::register('new/:id','New/read','GET|POST');
```

请求类型包括：

类型	描述
GET	表示GET请求

类型	描述
POST	表示POST请求
PUT	表示PUT请求
DELETE	表示DELETE请求
*	表示任何请求类型

系统提供了不同的请求类型简化方法，例如：

```
// 定义GET请求路由规则
\Think\Route::get('路由规则','路由地址','路由参数','变量规则');
// 定义POST请求路由规则
\Think\Route::post('路由规则','路由地址','路由参数','变量规则');
// 定义PUT请求路由规则
\Think\Route::put('路由规则','路由地址','路由参数','变量规则');
// 定义DELETE请求路由规则
\Think\Route::delete('路由规则','路由地址','路由参数','变量规则');
// 所有请求都支持的路由规则
\Think\Route::any('路由规则','路由地址','路由参数','变量规则');
```

例如：

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

闭包路由的增强

闭包路由可以进行路由劫持或者中断处理，如果返回的是一个数组，那么表示路由到数组表示的模块、控制器和操作继续进行，例如：

```
\Think\Route::get('new/:id',function($id){
    // 这里进行必要的处理
    // 返回路由信息
    return ['index','new','read'];
}); // 定义GET请求路由规则
```

> 如果直接输出或者返回字符串变量的话，闭包方法执行后会终止执行后续的操作。

路由参数

路由参数主要用于验证当前的路由规则是否有效，支持请求类型（method）、URL后缀（ext）、HTTPS

请求（https）以及自定义检测机制（callback），使用方法：

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

变量规则

ThinkPHP5.0支持在规则路由中为变量用正则的方式指定变量规则，弥补了之前变量规则太局限的问题，并且支持全局规则设置。使用方式如下：

设置全局变量规则，全局路由有效：

```
// 设置name变量的正则规则
\Think\Route::pattern('name','\w+');
// 支持批量添加
\Think\Route::pattern(['name'=>'\w+','id'=>'\d+']);
```

局部变量规则，仅在当前路由有效：

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

如果一个变量同时定义了全局规则和局部规则，局部规则会覆盖全局变量的定义。

域名路由

可以给（子）域名或者IP指定路由规则，例如：

```
// blog.thinkphp.cn 路由到index模块的blog控制器
\Think\Route::domain('blog.thinkphp.cn','index/blog');
// admin.thinkphp.cn 路由到admin模块
\Think\Route::domain('admin.thinkphp.cn','admin');
```

路由分组

路由定义多了之后效率会变低，这个时候可以使用路由分组功能，把相同前缀的路由规则定义到一个分组，并且为分组定义相同的路由参数。

```
\Think\Route::group('new',[':id'=>'New/read',':name'=>'New/read'],'GET',['ext'=>'html',['name'=>'\w
+','id'=>'\d+']);
// 相当于同时注册以下路由
\Think\Route::register('new/:id','New/read','GET',['ext'=>'html',['id'=>'\d+']);
\Think\Route::register('new/:name','New/read','GET',['ext'=>'html',['name'=>'\w+']);
```

更灵活的控制器

新版的控制器设计的更为灵活，主要体现在如下几个方面：

- 可以无需继承任何控制器类；
- 控制器操作方法中无需关心输出问题；
- 直接支持在操作方法中操作视图类；
- 可以通过引入Traits或者继承扩展类的方式来增加控制器功能；
- 支持任意层次的控制器类设计；

新的控制器输出和渲染方式

新的控制器输出方式

新版的控制器设计本身和视图类以及模型类没有任何的绑定，操作方法中只需要返回需要输出的数据（包括字符串和数组，甚至是对象），例如：

```
namespace app\index\controller;

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

这是一个典型的控制器类和操作方法定义，虽然我们也可以采用下面的方式，但并不建议：

```
namespace app\index\controller;

class Index {
    public function index(){
        echo 'hello,world!';
    }
}
```

采用前面一种返回数据的方式有利于你根据需要输出不同的格式，以及进行统一的拦截处理。

> 控制器的输出由系统的\Think\Response类接管了，并且由default_return_type参数配置决定输出的格式，包括json、xml等。

模板渲染

如果你需要在控制器的操作方法中渲染模板进行输出，可以使用下面两种方式：

一、继承系统封装的\Think\Controller类

```
namespace app\index\controller;

use think\Controller;

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

fetch('hello')方法默认会读取下面的模板文件渲染输出：

```
application/index/view/index/hello.html
```

如果使用空的fetch()方法的话则会渲染输出：

```
application/index/view/index/index.html
```

如果你不是渲染模板，而是直接渲染内容解析模板标签进行输出的话，可以使用show方法，例如：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller {
    public function index(){
        $this->assign('name','thinkphp');
        return $this->show('hello,{$name}');
    }
}
```

二、直接实例化View类

我们把前面的例子改成直接实例化View类的方式如下：

```
namespace app\index\controller;

class Index {
    public function index(){
        $View = new \Think\View();
        $View->assign('name','thinkphp');
        return $View->fetch('hello');
    }
}
```

三、引入Traits的方式扩展

新版的ThinkPHP大量采用了Traits的方式进行扩展和引入，这对功能的组装带来极大的便利，同样，我们把上面的示例改造为引入traits的方式：

```
namespace app\index\controller;

// 引入Traits类（PHP5.5以上版本可以不需要）
T('controller/View');
class Index {

    use \traits\controller\View;
    public function index(){
        $this->assign('name','thinkphp');
        return $this->fetch('hello');
    }
}
```

引入Traits的优势是你可以同时引入多个Traits，不受PHP类只能继承一个的限制，例如：

```
namespace app\index\controller;

// 引入Traits类（PHP5.5以上版本可以不需要）
T('controller/View');
T('controller/Jump');
class Index {

    use \traits\controller\View;
    use \traits\controller\Jump;
    public function index(){
        if(!IS_AJAX){
            $this->error('非法操作');
        }else{
            $this->assign('name','thinkphp');
            return $this->fetch('hello');
        }
    }
}
```

控制器层次

新版的控制器可以支持任意层次的设计，并且没有相同层次的限制，例如下面定义了一个index\controller\one\two\Index 控制器类：

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

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

我们在访问该控制器的时候可以使用

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

当然，也可以用路由简化URL，如下：

```
\Think\Route::register('hello','index/one.two.index/index');
```

API开发支持

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。

```
namespace app\index\controller;

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

访问该请求URL地址后，最终可以在浏览器中看到输出结果如下：

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

> 如果你需要返回其他的数据格式的话，控制器本身的代码无需做任何改变。

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
\think\Hook::add('return_data','\app\index\behavior\DataToXml');
// 或者直接使用闭包函数处理
\think\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通过整合SocketLog可以实现远程的开发调试。
设置方式：

```
'log'          => [
    'type'          => 'socket',
    // socket服务器
    'host'          => '111.202.76.133',
],
```

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。

内置SocketLog

自ThinkPHP5.0开始，框架内部整合了SocketLog的本地/远程调试方法。

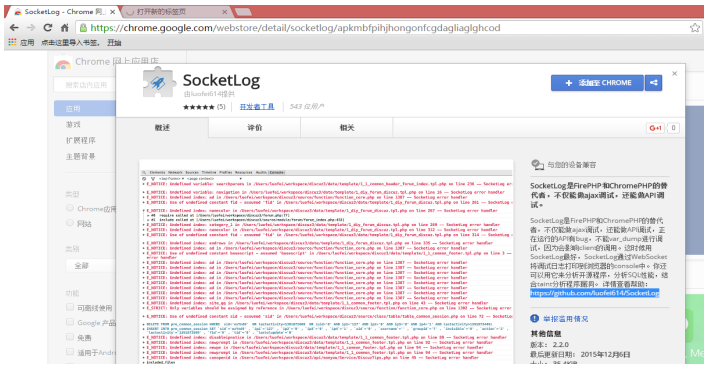
请注意，调试前，你需要安装chrome插件。

一、安装Chrome插件

1、访问插件主页（需要翻墙）

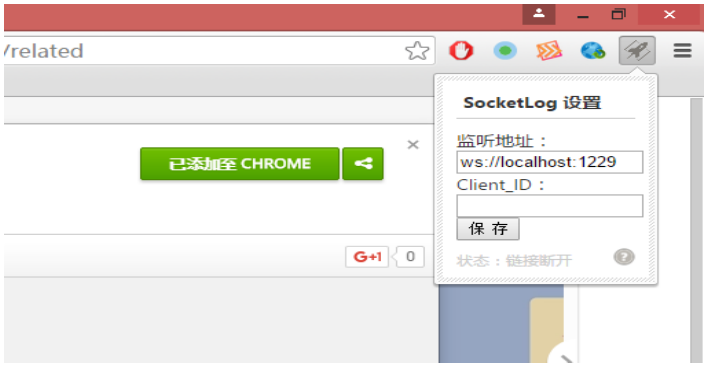
https://chrome.google.com/webstore/detail/socketlog/apkmbfpihjhongonfcgdagliaglghcod

翻墙后访问上面的链接，点击“添加至CHROME”



2、插件配置

插件安装成功后，你可以在浏览器右上角找到插件图标，点击配置。



> 配置时，注意监听地址，如果localhost则表示本地已经安装配置服务端
> Client_ID 相当于协定好的秘钥，会在下面的框架配置中提到。

二、框架配置

1、修改配置文件

添加如下代码：

```
'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'    => [],
    //上面的client_id,就是对应上面插件中设置的Client_ID
    //一个是推送，一个是限制
]
```

请注意，采用socket的方式调试会和Trace调试冲突，你只能选择一个

三、安装服务端（可选）

1、安装service

如果你安装过nodejs，并配置成功；你可以采用下面的方法直接安装：
运行 `npm install -g socketlog-server` 直接安装
运行 `socketlog-server` 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 &`

> 同时作者提供了公用服务端，可以前往 http://slog.thinkphp.cn/ 申请client_id。

四、GitHub主页

https://github.com/luofei614/SocketLog

新的自动生成

利用Traits特性扩展

自动加载及Loader类

命令行工具集

附录

升级指南

我以自己的oneblog体验了一下tp5 beta 版的使用，写下最小修改的升级方式。

目的

告诉大家如何去移植旧项目到新版tp5 beta中去。以及可能遇到的问题，和老杨是怎么解决的。

前提

做正确的移植之前，我们得熟读一下本手册，这样才能少走弯路。

几个巨大的变化

命名规范

一开始说是PSR0 后来大家协作时说规范是PSR2， 有人建议Sublime 装一个phpfmt 插件，可是我装了提示php版本低于5.6 用不了。不折腾了。
建议大家看看之前的命名规范章节。 主要是命名空间，目录和文件名小写，分割用_，但是类名要大写首字母驼峰式。开始老杨也不理解，为什么类名不也小写。后来一想，类名一小写。就难和文件和目录区分开了。

配置不合并

以往我们记得有个Common模块， 里面有公共函数、公共配置，然后其他模块的会先加载这个配置和函数然后去跟当前模块的合并。现在不行了。现在的公共配置不放在Common里，放在application APP_PATH根目录下。什么config.php、database.php、common.php啦。后来问老大那现在的Common 干嘛用，他说放公共的类，比如公共的model、behavior

Traits

新框架用了php5.4开始添加的Traits功能，老杨看了下，就是用于多继承的，使得代码更精简，比方 原来我们有高级模型、视图模型，然后，我们一个类既有视图模型、又有高级模型的功能，要实现这样的怎么办，定义一个第三方模型，复制这两个模型里的代码。。 现在有了Traits，我们可以 通过use 关键字，将Traits 实现的功能，直接在模型里 复用一下。框架新增了T函数是为了兼容php5.4版本的，5.5的可以直接引用。

因此，老大在架构上把一些常用的自动完成、高级模型给分离开到traits目录里去了，这带来了一些不便。后面我会讲如何去使用，达到以前的效果。

调试

以前的调试工具条被舍弃了，换成了专门调试api的不影响页面输出的SocketLog工具。当然也不说不能用，只是有一些不便：依赖网络、空值不输出。

耦合低了，很多东西独立开来了，比如视图 以前我们控制器里可以直接display、error、succes。现在必须依赖视图类，因为老大认为面向api的框架，很少需要视图。当然如果能用视图的话，原先的视图功能还是完整的，只不过使用上不方便。

移植的步骤

入口+框架

首先去https://github.com/top-think/think git 工具下载一份 beta版 tp5。 放入自己的环境里去。

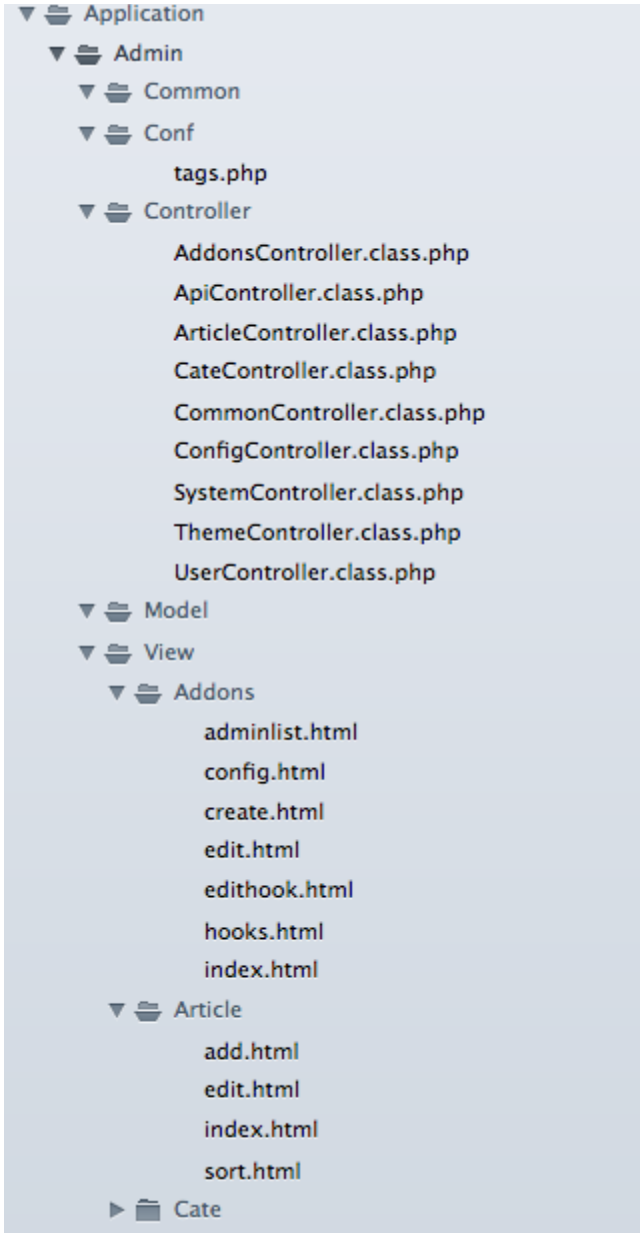
然后修改入口文件，改为以下的：

```
<?php
// +----------------------------------------------------------------------
// | ThinkPHP [ WE CAN DO IT JUST THINK ]
// +----------------------------------------------------------------------
// | Copyright (c) 2006-2015 http://thinkphp.cn All rights reserved.
// +----------------------------------------------------------------------
// | Licensed ( http://www.apache.org/licenses/LICENSE-2.0 )
// +----------------------------------------------------------------------
// | Author: liu21st <liu21st@gmail.com>
// +----------------------------------------------------------------------

// 应用入口文件
// 定义项目路径
define('APP_PATH', './application/');
define('ONETHINK_ADDON_PATH', './addons/');
// 开启调试模式
define('APP_DEBUG', true);
define('APP_HOOK', true);
define('SLOG_ON', true);

// define('BIND_MODULE','home');
// define('BIND_CONTROLLER','index');

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

自动生成

默认的示列应用APP_PATH下带了一个build.php 自动生成配置文件，我们可以修改一下，用于生成自己以前项目的目录结构，不过有点繁琐。

比方老杨以前的tp3.2的application下的结构是这样子的：



添加自动完成时，一个模块一个模块的加。然后依次是模块对应的子目录 __dir__ 、 __file__ 、controller、model、view 之类的子目录。比如老杨的admin模块：

```
'admin' => [
    '__file__'   => [],
    '__dir__'    => ['controller', 'view'],
    'controller' => ['Addons', 'Article', 'Cate', 'Common', 'Config', 'System', 'Theme', 'User'],
    'view'       => ['addons/index', 'article/add', 'cate/add', 'config/index', 'public/base', 'system/index', '
tags/index', 'theme/index', 'user/index'],
],
```

因为模块里的视图模板文件太多了，一个个手写很累，所以老杨只写了一个控制器里的一个模板，生成好后，手动复制过来整个目录的模板即可（因为模板一般都小写的，不用改文件名）。

自动生成控制器和模型的目的是节省时间，因为变化太大了：没有.class.php后缀，首字母要小写，对应命名空间也变了。能偷懒为什么不用呢。其实如果针对移植升级，可以写一个小脚本，扫描旧目录去生成对应新版的。

PS：sublime 对于rename这件事有个bug，就是如果有的目录或者文件更改大小写，然后这个目录和文件就打不开了。即使用菜单的刷新功能更也没用。只能手动关闭整个项目，再重新打开st。老杨发现重命名时先改小写的，同时多加一个s后 先重命名一次，然后在去掉s 百分百不会触发bug。

PS：自动生成每次访问都会生成，如果你后来想删除某些文件了。记得把自动生成的也改一下，要是node可以用gulp之类的watch 一下。php 还是人工吧。

完整的oneblog 生成：

```
return [
    // 生成运行时目录
    'runtime' => [
        '__dir__' => ['cache', 'log', 'temp'],
    ],
    'admin' => [
        '__file__'   => [],
        '__dir__'    => ['controller', 'view'],
        'controller' => ['Addons', 'Article', 'Cate', 'Common', 'Config', 'System', 'Theme', 'User'],
        'view'       => ['addons/index', 'article/add', 'cate/add', 'config/index', 'public/base', 'system/index
', 'tags/index', 'theme/index', 'user/index'],
    ],
    'common'=>[
        '__file__'   => [],
        '__dir__'    => ['behavior', 'controller', 'model', 'api'],
        'controller' => ['Addon'],
        'model'      => ['Addons', 'Article', 'Cate', 'Config', 'File', 'Hooks', 'Picture', 'Tree', 'Url'],
    ],
    'home'=>[
        '__file__'   => ['config.php', 'common.php'],
        '__dir__'    => ['widget', 'controller', 'view'],
        'controller' => ['Addons', 'Api', 'Error', 'Index'],
        'view'       => ['index/index', 'widget/archive'],
        'widget'    =>['Common']
    ],
    // 。。。 其他更多的模块定义
];
```

更新数据库配置、slog和其他配置

首先数据库的配置独立出来再APP_PATH下的database.php。记得新版 咱数组用[] 来写，多精简。
然后就是老几项了。hostname、username、password 之类的。

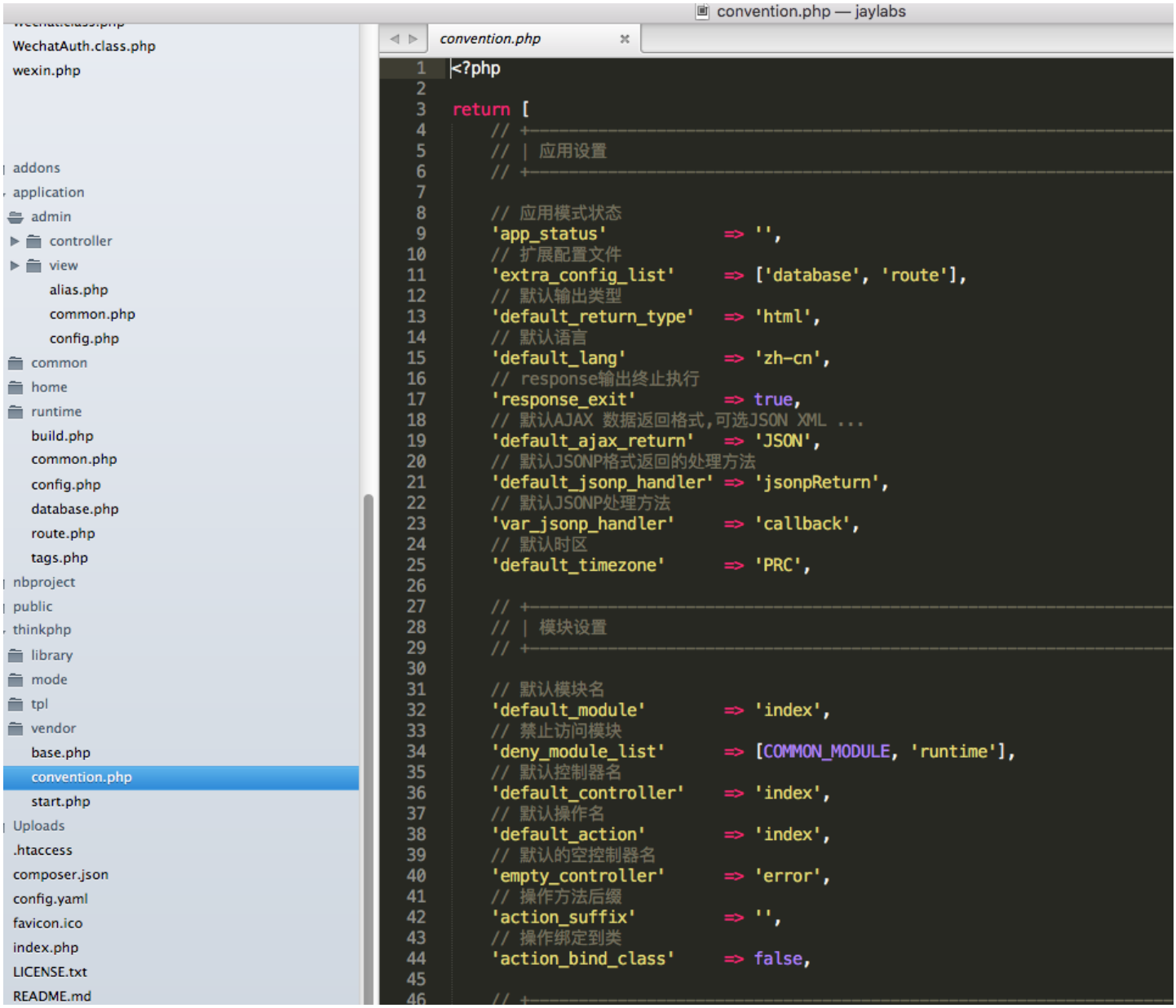
slog 的配置:

```
'slog'                  => [
        'enable'            => true, //是否记录日志的开关
        'host'              => '111.202.76.133',
        //是否显示利于优化的参数，如果允许时间，消耗内存等
        'optimize'          => true,
        'show_included_files' => true,
        'error_handler'     => true,
        //日志强制记录到配置的client_id
        'force_client_id'   => 'XXX',
        //限制允许读取日志的client_id
        'allow_client_ids'  => ['XXX'],
    ],
```

新版的配置文件中键名都是小写。老的用来写旧版的和自定义配置吧。

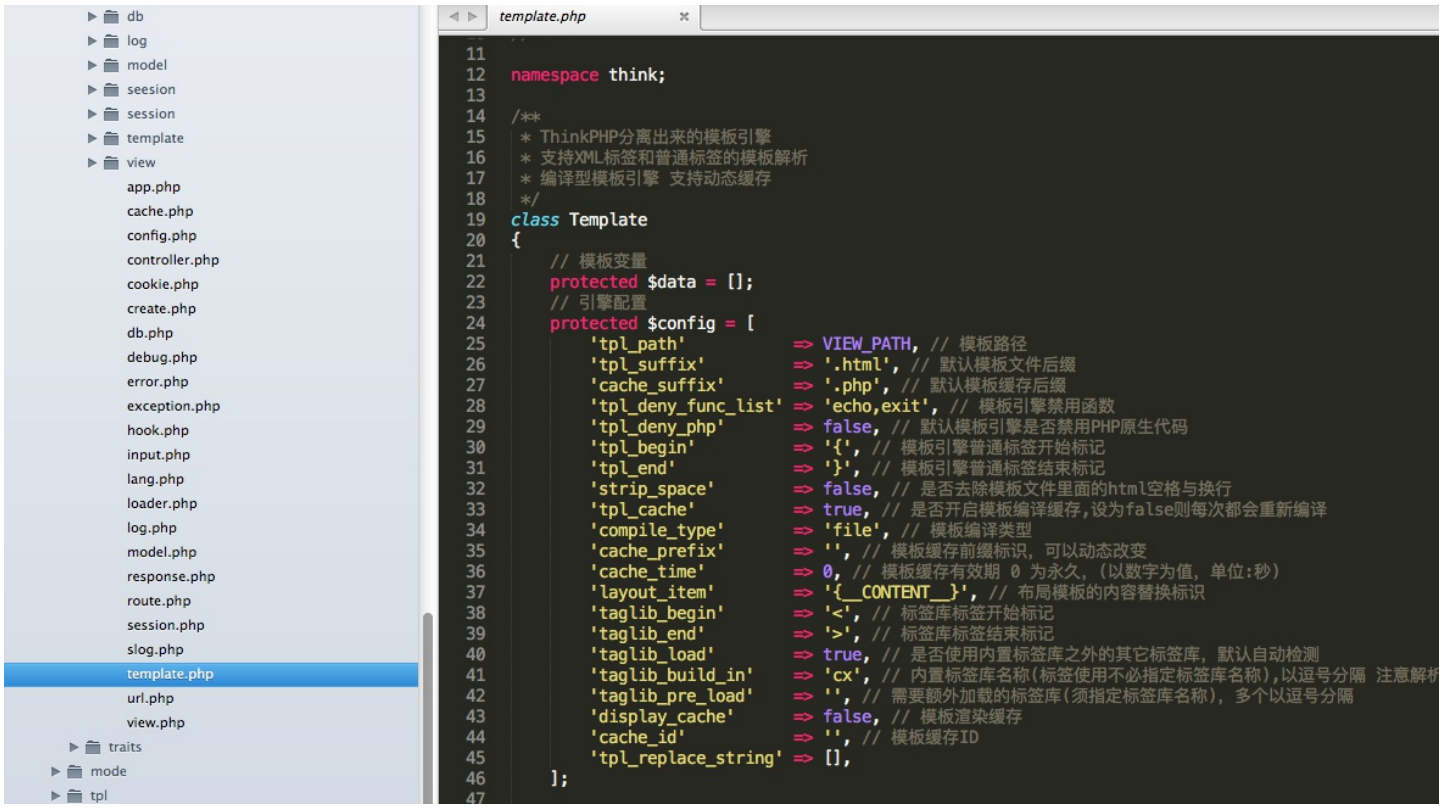
注意的是入口要定义 SLOG_ON 常量为true才起作用，老杨也是追了源码才知道的。

其他配置项，可以参考框架的convention.php文件



有对应的注释。

PS：有的类的构造方法里的配置，需要写在对应的命名空间里，如



template的配置，并不是不可变，而是要

```
'template'=>[
    'compile_type'=>'sae'
]
```

这样去改变，老杨也是测试sae试出来的。

移植模块目录里的函数、配置、tags

公共函数移出来，对应模块的函数复制到对应模块的common.php中，配置也是对应的config.php tags.php也是，只不过现在没有多余的Common和Config目录了，都是单文件。

将视图复制过来，修复命名上的错误

将对应的模块的view目录复制过来。基本上就能用了，但是注意一件事，include 和 extend 的name 对应的路径，只支持控制器/模板名了，且区分大小写。因此Public/Common 和public::common都是不对的。必须public/common
等搞定视图输出，时测试一些用法。老杨测试出include 属性变量没解析对和switch的 case 多层嵌套有问题，当然已经修复了，你们可以测测看有什么不兼容的。去提issue。

局部访问 测试bug 差异化

控制器的一些方法、框架函数之类的。
常见单字母函数，框架还是留了可以兼容。如D('Article') 会找当前模块的，找不到去找common/model下的article.php就是Article类。

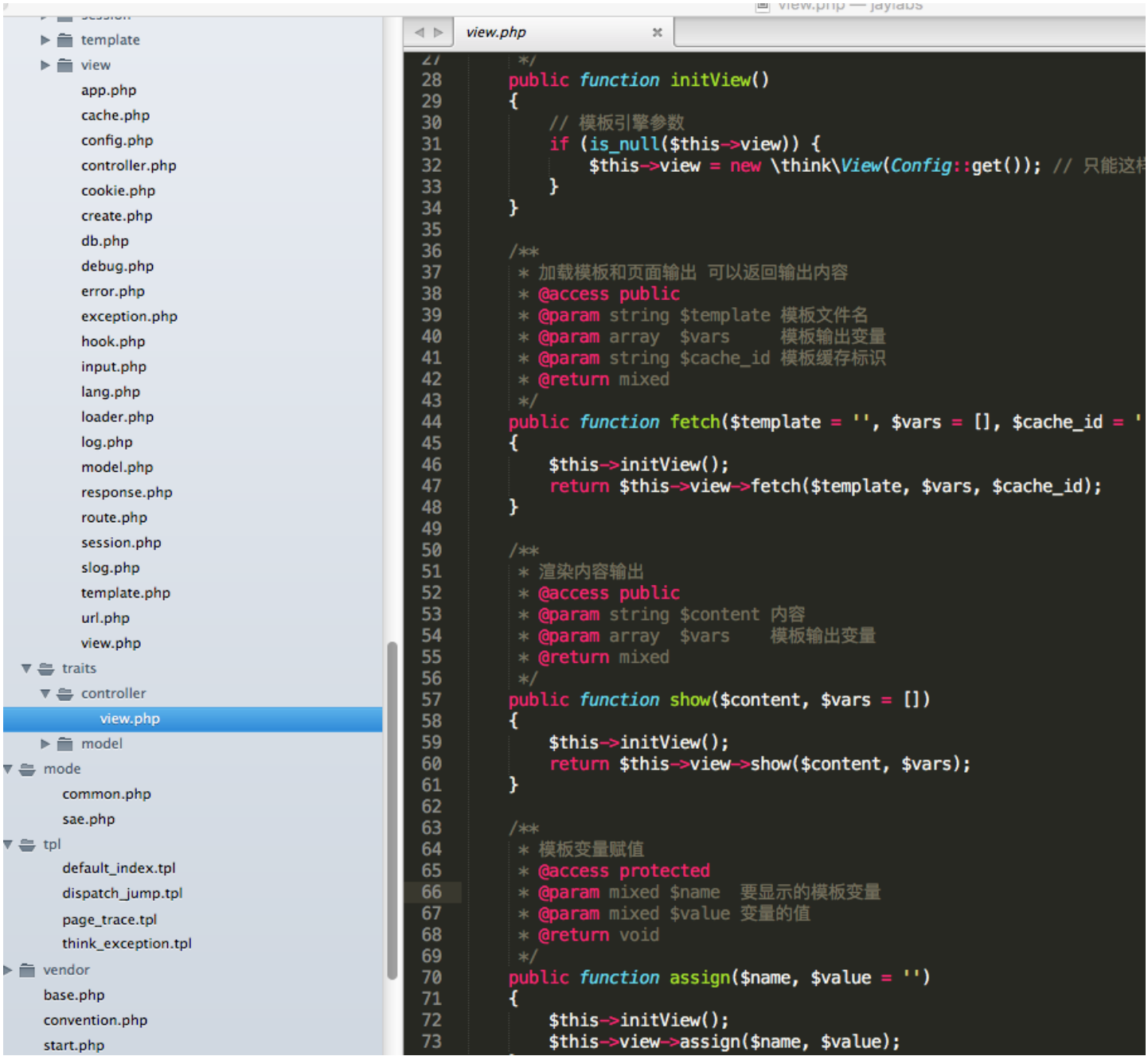
遇到的问题

公共模块的配置不合并了？

参见上面的介绍，位置变了。

视图里一些方法不见了

为了能更好的测试，及系统流程的变化，现在tp的控制器必须有返回值。（特殊方法除外）。
返回给Response类（去thinkphp\libary\think下找），
所以官方的例子是 new \think\View 自己fetch 自己return。
后来群友提出来以前方便的succes和error 自动判断ajax 不见。我觉得每次自己实例化视图类很麻烦，老大就用Traits支持了 controller类里 的视图替换字符串变了fetch、show、assign 方法：



因此我们只需要有视图的控制器继承 \think\controller类就行了和之前一样的assign、show、及抛弃

display 用 fetch 替代。return 模板字符串。

配置精简了

大家看convention.php 文件就知道配置少了很多了。因为很多行为也没了。

分页类没有了

为了内核的精简，老大没有加，老杨手动移植了一个放到org和ot里 ot 自定义了样式。其他的类你看着放吧！

$this->redirect 没了

去response类看看 可以 `\think\response::redirect`

模型默认很多快捷操作没有了？

这个很头痛，比方说以前M('article')->count()、 M('Article')->getField 和getFieldBy 都不能用了。

现在只能 D后用，然后在对应的模型里，使用 traits。

如

```
<?php
namespace common\model;
T('model/adv');
T('model/auto');
class Article extends \think\Model{
    use \traits\model\adv;
    use \traits\model\auto;
    public function _initialize(){
        /* 自动验证规则 */
        $this->validate = array(
            array('name', '/^[a-zA-Z]\w{0,39}$/', '文档标识不合法', self::VALUE_VALIDATE, 'regex', self::MOD
EL_BOTH),
            array('name', '', '标识已经存在', self::VALUE_VALIDATE,'unique'),
            array('title', 'require', '标题不能为空', self::MUST_VALIDATE, 'regex', self::MODEL_BOTH),
            array('title', '1,80', '标题长度不能超过80个字符', self::MUST_VALIDATE, 'length', self::MODEL_BOT
H),
            array('description', '1,140', '简介长度不能超过140个字符', self::VALUE_VALIDATE, 'length', self::M
ODEL_BOTH),
            array('cate_id', 'require', '分类不能为空', self::MUST_VALIDATE , 'regex', self::MODEL_INSERT),
            array('cate_id', 'require', '分类不能为空', self::EXISTS_VALIDATE , 'regex', self::MODEL_UPDATE),
        );

        /* 自动完成规则 */
        $this->auto = array(
            array('uid', 'is_login', self::MODEL_INSERT, 'function'),
            array('title', 'htmlspecialchars', self::MODEL_BOTH, 'function'),
            array('content', 'base_encode', self::MODEL_BOTH, 'callback'),
            array('description', 'htmlspecialchars', self::MODEL_BOTH, 'function'),
            array('link_id', 'getLink', self::MODEL_BOTH, 'callback'),
            array('view', 0, self::MODEL_INSERT, 'string'),
            array('comment', 0, self::MODEL_INSERT, 'string'),
            array('create_time', 'getCreateTime', self::MODEL_BOTH,'callback'),
            array('update_time', NOW_TIME, self::MODEL_BOTH, 'string'),
            array('status', '1', self::MODEL_INSERT, 'string'),
        );
    }
```

有自动完成和自动验证的用 traits/auto。 有after 后置操作，和快捷操作的 用 traits/adv。 发现自己的查询不支持了，先去 traits/model 里找找。

因此有时为了一个简单的复杂查询得建一个简单模型：
如user

```
<?php
namespace common\model;
T('model/adv');
class User extends \Think\Model{
    use \traits\model\adv;
}
```

注意的是新版的 auto 和 valiate 不带_ 了，我的做法是 _initialize 方法里 属性赋值或者直接= 以前的数组。

模板路径替换不能用了？

因为新版 现在没有任何内置行为。
开始我是移植旧的ContentReplace行为。 `parse_str` 这个配置。

配置里将以前的TPML_PARSE_STRING 替换成这个键名即可。

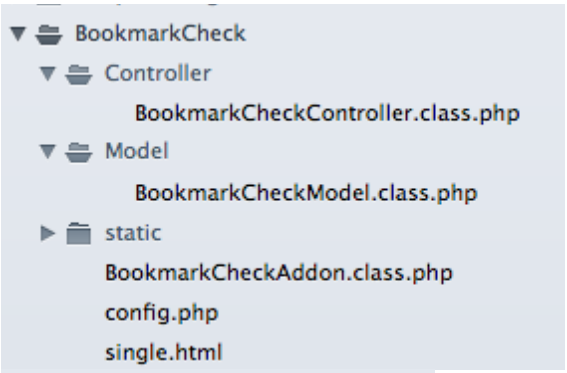
值的注意的是 以前3.2版 dispatch 里定义的很多常量没了， `__ROOT__` 之类的，我挑了3个常用的 `__ROOT__` 、 `__APP__` 、 `__SELF__`

```
if (!IS_CLI) {
    // 当前文件名
    if (!defined('_PHP_FILE_')) {
        if (IS_CGI) {
            //CGI/FASTCGI模式下
            $_temp = explode('.php', $_SERVER['PHP_SELF']);
            define('_PHP_FILE_', rtrim(str_replace($_SERVER['HTTP_HOST'], '', $_temp[0] . '.php'), '/'));
        } else {
            define('_PHP_FILE_', rtrim($_SERVER['SCRIPT_NAME'], '/'));
        }
    }
    if (!defined('__ROOT__')) {
        $_root = rtrim(dirname(_PHP_FILE_), '/');
        define('__ROOT__', (($_root == '/' || $_root == '\\') ? '' : $_root));
    }
    define('PHP_FILE', _PHP_FILE_);
}
if(!defined('__APP__'))
    define('__APP__', strip_tags(PHP_FILE));
// URL常量
if(!defined('__SELF__'))
    define('__SELF__', strip_tags($_SERVER[C('URL_REQUEST_URI')]));
```

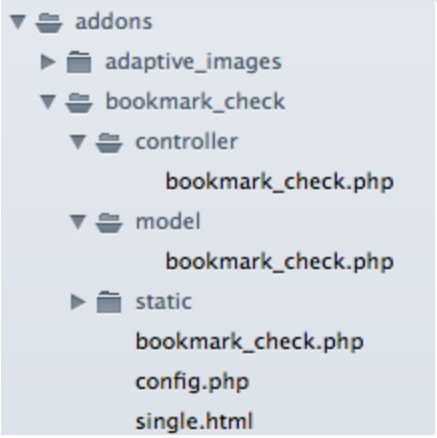
将旧版的移植到 config.php return 上面。本来是想放common.php公共函数里的，结果不起作用。

插件不能用了？

由于oneblog 移植了ot的插件机制，所以也得研究一下了。
插件依赖钩子，所以入口要定义HOOK_ON 常量。
然后头疼的就是命名空间了。
3.2的插件目录结构和文件名：



5.0 的：



就是插件目录小写了，后缀变了。目录名也变了，然后类名不变。

以前一个控制器是BookMark/BookMark 现在变成了 book_mark/BookMark！！！

然后咱找个地方定义插件目录常量 入口：

`define('ONETHINK_ADDON_PATH', './addons/');`

添加命名空间：

Think\Loader::addNamespace('addons', ONETHINK_ADDON_PATH);

公共函数里添加命名空间印射目录。

然后修改common/behavior/init_hook.php 初始化钩子行为

```
<?php

// +----------------------------------------------------------------------
// | ThinkPHP [ WE CAN DO IT JUST THINK IT ]
// +----------------------------------------------------------------------
// | Copyright (c) 2006-2013 http://thinkphp.cn All rights reserved.
// +----------------------------------------------------------------------
// | Licensed ( http://www.apache.org/licenses/LICENSE-2.0 )
// +----------------------------------------------------------------------
// | Author: liu21st <liu21st@gmail.com>
// +----------------------------------------------------------------------

namespace common\behavior;
use think\Hook;
use think\Loader;
defined('THINK_PATH') or exit();

// 初始化钩子信息
class InitHook{

    // 行为扩展的执行入口必须是run
```

```
    // 行为扩展的执行入口必须是run
    public function run(&$content) {
        if (isset($_GET['m']) && $_GET['m'] === 'Install')
            return;

        // $data = null;
        $data = S('hooks');
        if (!$data) {
            $hooks = D('Hooks')->getField('name,addons');
            foreach ($hooks as $hook => $value) {
                if ($value) {
                    $map['status'] = 1;
                    $names = explode(',', $value);
                    $map['name'] = array('IN', $names);
                    $data = D('Addons')->where($map)->getField('id,name');
                    if ($data) {
                        $addons = array_intersect($names, $data);
                        foreach ($addons as $key => $value) {
                            $dir = Loader::parseName(lcfirst($value));
                            $path = "./addons/{$dir}/".Loader::parseName(lcfirst($value)).".php";
                            $value = "addons\\{$dir}\\{$value}";
                            $path = realpath($path);
                            $addons[$key] = $value;
                            Loader::addMap($value, $path);
                        }
                        Hook::add($hook, $addons);
                    }
                }
            }
            S('hooks', Hook::get());
        } else {
            Hook::import($data, false);
        }
    }

}
```

里面Loader::addMap 可是我实践出来的，开始以为定义了命名空间能自已找到类，结果得添加alias。然后就是公共tags里添加行为：

```
<?php

return [
    'app_init' => ['common\\behavior\\InitHook'],
];
```

这里写的都是命名空间规则，而不是路径。

common/controller/addon.php 插件定义抽象类修改，改getName 获取路径等方法，改着改着就成了这样：

```
<?php
```

```php
<?php
namespace common\controller;

/**
 * 插件类
 * @author yangweijie <yangweijiester@gmail.com>
 */
abstract class Addon extends \think\controller{

    /**
     * 视图实例对象
     * @var view
     * @access protected
     */
    protected $view = null;

    /**
     * $info = array(
     *  'name'=>'Editor',
     *  'title'=>'编辑器',
     *  'description'=>'用于增强整站长文本的输入和显示',
     *  'status'=>1,
     *  'author'=>'thinkphp',
     *  'version'=>'0.1'
     *  )
     */
    public $info = array();
    public $addon_path = '';
    public $config_file = '';
    public $custom_config = '';
    public $admin_list = array();
    public $custom_adminlist = '';
    public $access_url = array();

    public function __construct() {
        $this->addon_path = ONETHINK_ADDON_PATH . lcfirst($this->getName()) . '/';
        $parse_str = C('parse_str');
        $parse_str['__ADDONROOT__'] = __ROOT__ . '/addons/' . $this->getName();
        C('parse_str', $parse_str);
        if (is_file($this->addon_path . 'config.php')) {
            $this->config_file = $this->addon_path . 'config.php';
        }
    }

    /**
     * 模板主题设置
     * @access protected
     * @param string $theme 模版主题
     * @return Action
     */
    final protected function theme($theme) {
        $this->view->theme($theme);
        return $this;
    }
    final protected function addon_path(){
```

```
        return ONETHINK_ADDON_PATH . lcfirst($this->getName()) . '/';
    }

    //显示方法
    final protected function display($template = '') {
        if (!is_file($template)) {
            $template = $this->addon_path() . $template . '.html';
            if (!is_file($template)) {
                throw new \Exception("模板不存在:$template");
            }
        }
        $html = $this->fetch($template);
        return $html;
    }

    /**
     * 获取插件目录或名称
     * @param $type 0 - dir 1- name
     */
    final public function getName($type = 0) {
        $class = get_class($this);
        $class = str_replace("addons\\", '', $class);
        $class_arr = explode('\\', $class);
        return $class_arr[$type];
    }

    final public function checkInfo() {
        $info_check_keys = array('name', 'title', 'description', 'status', 'author', 'version');
        foreach ($info_check_keys as $value) {
            if (!array_key_exists($value, $this->info))
                return FALSE;
        }
        return TRUE;
    }

    /**
     * 获取插件的配置数组
     */
    final public function getConfig($name = '') {
        static $_config = array();
        if (empty($name)) {
            $name = $this->getName(1);
        }
        if (isset($_config[$name])) {
            return $_config[$name];
        }
        $config = array();
        $map['name'] = $name;
        $map['status'] = 1;
        $config = D('Addons')->where($map)->getField('config');
        if ($config) {
            $config = json_decode($config, true);
            return $config;
        } else if(!empty($this->config_file)){
            $temp_arr = include $this->config_file;
```

```
            $temp_arr = include $this->config_file;
            foreach ($temp_arr as $key => $value) {
                if ($value['type'] == 'group') {
                    foreach ($value['options'] as $gkey => $gvalue) {
                        foreach ($gvalue['options'] as $ikey => $ivalue) {
                            $config[$ikey] = $ivalue['value'];
                        }
                    }
                } else {
                    $config[$key] = $temp_arr[$key]['value'];
                }
            }
            $_config[$name] = $config;
            return $config;
        }else{
            return [];
        }
    }

    //必须实现安装
    abstract public function install();

    //必须卸载插件方法
    abstract public function uninstall();
}
```

后来就是改后台相关的类。主要改了显示的地方，config方法和controller的冲突了换成conf。后面框架会去掉这个方法。

home 和admin 插件控制器执行方法：

```
<?php
namespace home\controller;

class Addons extends \think\controller{
    protected $addons = null;

    public function execute($_addons = null, $_controller = null, $_action = null){
        $dir = \think\Loader::parseName($_addons, 0);
        $_controller = \think\Loader::parseName($_controller,1);
        $parse_str = C('parse_str');
        $parse_str['__ADDONROOT__'] = __ROOT__ . "/addons/{$dir}";
        C('parse_str', $parse_str);

        if(!empty($_addons) && !empty($_controller) && !empty($_action)){
            $addons = A("addons\\$dir/$dir")->$_action();
            return $addons;
        } else {
            return $this->error('没有指定插件名称，控制器或操作！');
        }
    }

    public function display($tpl = ''){
        if (!is_file($tpl)) {
            $tpl = $this->addon_path() . $tpl . '.html';
            if (!is_file($tpl)) {
                throw new \Exception("模板不存在:$tpl");
            }
        }
        $html = $this->fetch($tpl);
        return $html;
    }

    public function success($msg = '', $data = '', $url = '', $wait = 3){
        \think\Response::success($msg, $data, $url, $wait);
    }

    public function error($msg = '', $data = '', $url = '', $wait = 3){
        \think\Response::error($msg, $data, $url, $wait);
    }
}
```

PS:后台插件快速创建的还么改，大家可以自己参考类修改一下，因为这只是老杨自己的移植，并不是官方tp5 以后的插件架构。

值得注意的是插件模板的显示输出不要exit 也不要return 回给Hook::listen 里，因为， 一个钩子有多个插件，插件执行不一定都是返回模板。

所以，如果你要显示什么 请 @print($this->display('single')) 这样去写。

SAE上如何使用

think/model/sae 里的配置，要加入convetion.php里的部分配置。
如果你只是但应用，数据库就在当前应用，不需要建议database.php去配置数据库，如果是跨应用的，就要将来源引用的配置 常量 dump出来写死在项目公共数据库配置里。

然后是初始化一些服务 用于缓存的 memcache服务，用于文件上传的storage、注意sae的特性，不可写。

配置重定向要改 config.yaml
其他基本上没遇到问题。

尝试一下吧~

更多的以最新框架源码为准，只要没发布，每天更新仓库后自己手动覆盖已经保存svn或git的本地库，看看变化，自己的hack是否要去除了。
下载地址http://git.oschina.net/yangweijie/oneblog/tree/5.0beta/ 自己克隆了 切换分支后看readme。
尝鲜就得会折腾。

配置参考

错误编码

系统的错误信息编码对照表：

错误编码由5位数字组成：3位区块码+2位错误码。

错误对照表

编码	错误信息
10000	非法的控制器名称
10001	控制器类不存在
10002	控制器的操作方法不存在或者没有访问权限
10003	绑定到操作的类不存在
10004	URL参数绑定错误
10005	模块不存在或者禁止访问
10006	目录没有写权限无法完成自动生成

编码	错误信息
10007	实例化一个不存在的类
10300	模型方法的参数传入错误的数据类型
10500	数据库未知错误
10501	PDOException
10502	数据库参数绑定出错
10503	WHERE表达式错误
11700	Session handler错误
11500	SocketLog中传入了错误的数据库link
11600	模板中不允许使用php标签
11601	模板中的标签解析错误
11602	模板缓存文件写入错误
10700	模板文件不存在

错误区块

编码	所在类库
100	App/Loader/路由
101	控制器（核心）
102	控制器（扩展）
103	模型（核心）
104	模型（扩展）
105	数据库
106	数据库驱动
107	视图
108	视图驱动
109	缓存
110	缓存驱动
111	行为
112	配置
113	日志
114	日志驱动

编码	所在类库
115	SocketLog
116	模板引擎
117	Session驱动
200以上	扩展类库