ThinkPHP
SINCE 2006

为API开发而设计

ThinkPHP5.1 完全开发手册

目　录

序言

> 手册阅读须知：本手册仅针对ThinkPHP `5.1` 版本（使用左右键（ `<--` 和 `-->` ）翻页阅读）

> 十二载初心不改，你值得信赖的PHP框架。ThinkPHPV5.1新年献礼！祝大家2018新年快乐！

ThinkPHP是一个免费开源的，快速、简单的面向对象的轻量级PHP开发框架，是为了敏捷WEB应用开发和简化企业应用开发而诞生的。ThinkPHP从诞生以来一直秉承简洁实用的设计原则，在保持出色的性能和至简的代码的同时，也注重易用性。遵循 `Apache2` 开源许可协议发布，意味着你可以免费使用ThinkPHP，甚至允许把你基于ThinkPHP开发的应用开源或商业产品发布/销售。

> ThinkPHP `5.1` 在 `5.0` 的基础上对底层架构做了进一步的改进，引入新特性，并提升版本要求。另外一个事实是， `5.1` 版本看起来对开发者更加友好，表现在目录结构更直观、调试输出更直观和代码提示更直观。
>
> >ThinkPHP `5.1` 运行环境要求 `PHP5.6+` ，虽然不支持 `5.0` 的无缝升级，但升级过程并不复杂（请参考升级指导）， `5.1.*` 版本均支持无缝升级。

主要新特性：

- 引入容器和Facade支持
- 依赖注入完善和支持更多场景
- 重构的（对象化）路由
- 支持注解路由
- 跨域请求支持
- 配置和路由目录独立
- 取消系统常量
- 助手函数增强
- 类库别名机制

- 模型和数据库增强
- 验证类增强
- 模板引擎改进
- 支持 PSR-3 日志规范

版权申明

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

>

捐赠我们

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

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

用手机扫描进行支付宝捐赠

基础

安装

`ThinkPHP5.1`的环境要求如下：

> - PHP >= 5.6.0
> - PDO PHP Extension
> - MBstring PHP Extension

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

> 5.1版本开始，官网不再提供下载版本，请使用 `Composer` 或者 `git` 方式安装和更新。

Composer安装

`ThinkPHP5`支持使用`Composer`安装

> 如果还没有安装`Composer`，你可以按 这里提到 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。

由于众所周知的原因，国外的网站连接速度很慢。因此安装的时间可能会比较长，我们建议通过下面的方式使用国内镜像。

> 打开命令行窗口（windows用户）或控制台（Linux、Mac 用户）并执行如下命令：
>
> ```
> composer config -g repo.packagist composer https://packagist.phpcomposer.com
> ```

如果你是第一次安装的话，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
```

这里的tp5目录名你可以任意更改，执行完毕后，会在当前目录下的`tp5`子目录安装最新版本的ThinkPHP，这个目录就是我们后面会经常提到的应用根目录。

如果你之前已经安装过，那么切换到你的应用根目录下面，然后执行下面的命令进行更新：

```
composer update topthink/framework
```

> 更新操作会删除 `thinkphp` 目录重新下载安装新版本，但不会影响 `application` 目录，因此不要在核心框架目录添加任何应用代码和类库。

> 安装和更新命令所在的目录是不同的，更新必须在你的应用根目录下面执行

如果出现错误提示，请根据提示操作或者参考Composer中文文档。

一般情况下，`composer` 安装的是最新的稳定版本，不一定是最新版本，如果你需要安装实时更新的版本，可以安装 `5.1.x-dev` 版本。

```
composer create-project topthink/think=5.1.x-dev tp5
```

Git安装

也可以使用`git`版本库安装和更新，`ThinkPHP5.1`主要分为应用和核心两个仓库，主要包括：

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

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

首先克隆下载应用项目仓库

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

然后切换到 `tp5` 目录下面，再克隆核心框架仓库（注意目录名称不要改变）：

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

> 如果你访问 `github` 速度比较慢，可以考虑下面两个国内GIT仓库（国内仓库以稳定版本为主，不确保实时更新）：

[码云]

```
应用项目：https://git.oschina.net/liu21st/thinkphp5.git
核心框架：https://git.oschina.net/liu21st/framework.git
```

[Coding]

```
应用项目：https://git.coding.net/liu21st/thinkphp5.git
核心框架：https://git.coding.net/liu21st/framework.git
```

> 由于目前仓库默认分支还不是 `5.1` 版本，你需要切换到 `5.1` 分支（首先进入thinkphp目录后执行下面的命令）
>
> ```
> git checkout 5.1
> ```

两个仓库克隆完成后，就完成了 `ThinkPHP5.1` 的 `Git` 方式下载，如果需要更新核心框架的时候，只需要切换到 `thinkphp` 核心目录下面，然后执行：

```
git pull
```

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

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 `ThinkPHP5.1` 的安装！

> 实际部署中，应该是绑定域名访问到 `public` 目录，确保其它目录不在WEB目录下面。

开发规范

命名规范

`ThinkPHP5.1`遵循`PSR-2`命名规范和`PSR-4`自动加载规范，并且注意如下规范：

目录和文件

- 目录使用小写+下划线；
- 类库、函数文件统一以`.php`为后缀；
- 类的文件名均以命名空间定义，并且命名空间的路径和类库文件所在路径一致；
- 类文件采用驼峰法命名（首字母大写），其它文件采用小写+下划线命名；
- 类名和类文件名保持一致，统一采用驼峰法命名（首字母大写）；

函数和类、属性命名

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

常量和配置

- 常量以大写字母和下划线命名，例如`APP_PATH`；
- 配置参数以小写字母和下划线命名，例如`url_route_on`和`url_convert`；
- 环境变量定义使用大写字母和下划线命名，例如`APP_DEBUG`；

数据表和字段

- 数据表和字段采用小写加下划线方式命名，并注意字段名不要以下划线开头，例如`think_user`表和`user_name`字段，不建议使用驼峰和中文作为数据表及字段命名。

请理解并尽量遵循以上命名规范，可以减少在开发过程中出现不必要的错误。

> 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

目录结构

> 相对于 5.0 来说， 5.1 版本目录结构的主要变化是配置目录和路由定义目录独立出来，不再放入应用类库目录（并且不可更改）。

```
www  WEB部署目录（或者子目录）
├─application            应用目录
│   ├─common             公共模块目录（可以更改）
│   ├─module_name        模块目录
│   │   ├─common.php     模块函数文件
│   │   ├─controller     控制器目录
│   │   ├─model          模型目录
│   │   ├─view           视图目录
│   │   ├─config         配置目录
│   │   └─ ...           更多类库目录
│   │
│   ├─command.php        命令行定义文件
│   ├─common.php         公共函数文件
│   └─tags.php           应用行为扩展定义文件
│
├─config                 应用配置目录
│   ├─module_name        模块配置目录
│   │   ├─database.php   数据库配置
│   │   ├─cache          缓存配置
│   │   └─ ...
│   │
│   ├─app.php            应用配置
│   ├─cache.php          缓存配置
│   ├─cookie.php         Cookie配置
│   ├─database.php       数据库配置
│   ├─log.php            日志配置
│   ├─session.php        Session配置
│   ├─template.php       模板引擎配置
│   └─trace.php          Trace配置
│
├─route                  路由定义目录
│   ├─route.php          路由定义
│   └─...                更多
│
├─public                 WEB目录（对外访问目录）
│   ├─index.php          入口文件
│   ├─router.php         快速测试文件
│   └─.htaccess          用于apache的重写
│
├─thinkphp               框架系统目录
│   ├─lang               语言文件目录
│   ├─library            框架类库目录
```

```
│   │   ├─think            Think类库包目录
│   │   └─traits           系统Trait目录
│   │
│   ├─tpl                  系统模板目录
│   ├─base.php             基础定义文件
│   ├─convention.php       框架惯例配置文件
│   ├─helper.php           助手函数文件
│   └─logo.png             框架LOGO文件
│
├─extend                   扩展类库目录
├─runtime                  应用的运行时目录（可写，可定制）
├─vendor                   第三方类库目录（Composer依赖库）
├─build.php                自动生成定义文件（参考）
├─composer.json            composer 定义文件
├─LICENSE.txt              授权说明文件
├─README.md                README 文件
├─think                    命令行入口文件
```

> 在 mac 或者 linux 环境下面，注意需要设置 runtime 目录权限为777。

> 由于`5.1`版本取消了系统路径的常量定义，因此系统的目录名称不可更改。如果需要更改应用目录或者入口文件位置，参考架构章节的入口文件部分。

配置

配置基础

`ThinkPHP`遵循惯例重于配置的原则，系统会按照下面的顺序来加载配置文件（配置的优先顺序从右到左）。

> 惯例配置->应用配置->模块配置->动态配置

- 惯例配置：核心框架内置的配置文件，无需更改。
- 应用配置：每个应用的全局配置文件（框架安装后会生成初始的应用配置文件），有部分配置参数仅能在应用配置文件中设置。
- 模块配置：每个模块的配置文件（相同的配置参数会覆盖应用配置），有部分配置参数模块配置是无效的，因为已经使用过。
- 动态配置：主要是指在控制器或者行为中进行（动态）更改配置，该配置方式只在当次请求有效，因为不会保存到配置文件中。

> 和5.0最大的区别是，5.1版本的配置参数全部是二级配置，当没有指定一级配置名的时候，默认就是以 app 作为一级配置，一级配置名称通常来说就是所在的配置文件名。

配置文件和目录

为更好的应对模块化的开发要求，`5.1`的应用配置主要包括应用配置目录和模块配置目录，结构如下：

```
├─config（应用配置目录）
│   ├─app.php             应用配置
│   ├─cache.php           缓存配置
│   ├─cookie.php          Cookie配置
│   ├─database.php        数据库配置
│   ├─log.php             日志配置
│   ├─session.php         Session配置
│   ├─template.php        模板引擎配置
│   ├─trace.php           Trace配置
│   └─ ...                更多配置文件
│
├─route（路由目录）
│   ├─route.php           路由定义文件
│   └─ ...                更多路由定义文件
│
├─application （应用目录）
│   └─module （模块目录）
```

```
│      └─config（模块配置目录）
│          ├─app.php              应用配置
│          ├─cache.php            缓存配置
│          ├─cookie.php           Cookie配置
│          ├─database.php         数据库配置
│          ├─log.php              日志配置
│          ├─session.php          Session配置
│          ├─template.php         模板引擎配置
│          ├─trace.php            Trace配置
│          └─ ...                 更多配置文件
│
```

> 一定注意了，5.1没有 config.php 配置文件，默认配置都在 app.php 配置文件，并且配置参数区分大小写

上面的目录结构是只是列出系统内置的配置文件，你还可以增加其他的自定义配置文件，配置文件的名称就是一级配置名。

> 因为架构设计要求， 5.1 的路由配置文件（确切来说应该是路由定义文件）独立于配置文件（更多会在路由章节中讲述）。

应用和模块的配置目录对应关系为：

配置	目录
应用配置目录	config
模块配置目录	application/module/config

如果你需要统一管理所有的配置文件，那么可以把模块目录下面的 `config` 目录移动到应用配置目录下面改为模块子目录的方式，调整后的配置目录的结构如下：

```
├─application（应用目录）
├─config（配置目录）
│   ├─module （模块配置目录）
│   │   ├─database.php    数据库配置
│   │   ├─cache           缓存配置
│   │   └─ ...
│   │
│   ├─app.php             应用配置
│   ├─cache.php           缓存配置
│   ├─cookie.php          Cookie配置
│   ├─database.php        数据库配置
│   ├─log.php             日志配置
│   ├─session.php         Session配置
│   ├─template.php        模板引擎配置
```

```
│   └─trace.php           Trace配置
│
├─route（路由配置目录）
│   ├─route.php           路由定义文件
│   └─ ...                更多路由定义文件
```

> 一旦模块目录下面存在 config 目录，则应用配置目录下的模块配置则无效，也不会对两个目录的配置进行合并。

这样一来，配置目录对应关系变成：

配置	目录
应用配置目录	config
模块配置目录	config/module

每个配置文件中都有详细的配置参数说明，可以仔细查看。

配置定义

可以直接在相应的应用或模块配置文件中修改或者增加配置参数，如果你要增加额外的配置文件，直接放入应用或模块配置目录即可（文件名小写）。

> 除了一级配置外，配置参数名严格区分大小写，建议是使用小写定义配置参数的规范。

另外涉及到配置参数的定义有效性问题，下列配置参数在模块配置中定义（包括动态配置）无效，而必须在应用配置中设置：

配置参数	描述
app_debug	应用调试模式（支持环境变量配置）
app_trace	应用trace（支持环境变量配置）
class_suffix	类后缀
default_filter	默认过滤机制
root_namespace	根命名空间
pathinfo_depr	PATH_INFO分隔符
url_route_must	路由强制模式
auto_bind_module	自动绑定模块
default_lang	默认语言
lang_switch_on	多语言切换

由于架构设计原因，下面的配置只能在环境变量中修改。

配置参数	描述
app_namespace	应用命名空间
config_ext	配置文件后缀

其它配置格式支持

默认的配置文件都是PHP数组方式，如果你需要使用其它格式的配置文件，你可以通过改变 `CONFIG_EXT` 环境变量的方式来更改配置类型。

在应用根目录的 `.env` 或者系统环境变量中设置

```
CONFIG_EXT=".ini"
```

支持的配置类型包括 `.ini` 、 `.xml` 、 `.json` 、 `.yaml` 和 `.php` 在内的格式支持，配置后应用配置及模块配置必须统一使用相同的配置类型。

二级配置

配置参数的值同样支持数组，下面是示例：

```
return [
    'user'  =>  [
        'type'  =>  1,
        'name'  =>  'thinkphp',
    ],
    'db'    =>  [
        'type'      =>  'mysql',
        'user'      =>  'root',
        'password'  =>  '',
    ],
];
```

环境变量定义

可以在应用的根目录下定义一个特殊的 `.env` 环境变量文件，用于在开发过程中模拟环境变量配置（该文件建议在服务器部署的时候忽略）， `.env` 文件中的配置参数定义格式采用 `ini` 方式，例如：

```
APP_DEBUG =  true
APP_TRACE =  true
```

> 如果你的部署环境单独配置了环境变量（ 环境变量的前缀使用 `PHP_` ），那么请删除 `.env` 配置文件，避免冲突。

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

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

```
DATABASE_USERNAME =  root
DATABASE_PASSWORD =  123456
```

获取环境变量的值使用下面的方式：

```
Env::get('database_username');
Env::get('database_password');
```

如果使用

```
[DATABASE]
USERNAME =  root
PASSWORD =  123456
```

获取环境变量的值可以使用下面的两种方式获取：

```
Env::get('database.username');
Env::get('database.password');
```

要使用 `Env` 类，必须先引入 `think\facade\Env` 或者 `\Env` 。

> 环境变量的获取不区分大小写

可以支持默认值，例如：

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

可以直接在配置文件中使用环境变量进行本地环境和服务器的自动配置，例如：

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

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

配置获取

要使用 `Config` 类，首先需要在你的类文件中引入

```
use think\facade\Config;
```

或者（因为系统做了类库别名，其实就是调用 `think\facade\Config` ）

```
use Config;
```

然后就可以使用下面的方法读取某个配置参数的值：

```
echo Config::get('配置参数1');
```

如果你需要读取某个一级配置的所有配置参数，可以使用

```
Config::pull('app');
```

或者使用

```
Config::get('app.');
```

读取所有的配置参数：

```
dump(Config::get());
```

判断是否存在某个设置参数：

```
Config::has('配置参数2');
```

助手函数

系统定义了一个助手函数`config`，以上可以简化为：

```
echo config('配置参数1');
```

`5.1`的配置参数全部采用二级配置的方式（默认一级配置为app），所以当你使用`config('name')`的时候其实相当于使用：

```
config('app.name')
```

支持获取多级配置参数值，直接使用（必须从一级开始写）

```
config('app.name1.name2')
```

获取某个一级配置的所有参数可以使用

```
config('app.');
```

读取所有的配置参数：

```
dump(config());
```

或者你需要判断是否存在某个设置参数：

```
config('?配置参数2');
```

动态设置

在控制器或者行为里面可以使用`set`方法或者助手函数动态设置参数（不过需要注意的是，动态设置生效的前提是该参数尚未被使用），格式：

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

动态设置的参数，最多支持二级，例如：

```
Config::set('app_trace', true);
// 没有指定配置文件名的话 等效于下面的方式
Config::set('app.app_trace', true);
// 助手函数的方式
config('app_trace', true);
```

也可以传入数组批量设置，并在第二个参数传入一级配置名，例如：

```
Config::set([
    'app_trace'=>true,
    'show_error_msg'=>true
], 'app');

// 或者使用助手函数
config([
    'app_trace'=>true,
    'show_error_msg'=>true
],'app');
```

系统配置文件

下面系统自带的配置文件列表及其作用（配置文件可能同时存在应用配置文件和模块配置文件两个同名文件）：

配置文件名	描述
app.php	应用配置
cache.php	缓存配置
cookie.php	Cookie配置
database.php	数据库配置
log.php	日志配置
session.php	Session配置
template.php	模板引擎配置
trace.php	页面Trace配置
paginate.php	分页配置

具体的配置参数可以直接查看应用 `config` 目录下面的相关文件内容。

架构

架构总览

`ThinkPHP`支持传统的`MVC`（Model-View-Controller）模式以及流行的`MVVM`（Model-View-ViewModel）模式的应用开发，但无论采用何种模式，URL的规范仍然是统一的。

`5.1`的URL访问受路由决定，如果在没有定义或匹配路由的情况下（并且没有开启强制路由模式的话），则是基于：

> http://serverName/index.php（或者其它入口文件）/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`或者一个命令行程序入口`think`都属于入口文件。

应用

应用在`ThinkPHP`中是一个管理系统架构及生命周期的对象，由系统的`\think\App`类完成，应用通常在入口文件中被调用和执行，具有相同的应用目录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（绑定不同的模块或者使用不同的配置）。

应用具有自己独立的配置文件、公共（函数）文件和路由定义文件。

路由

路由是用于规划（一般同时也会进行简化）请求的访问地址，在访问地址和实际操作方法之间建立一个路由规则 => 路由地址的映射关系。

ThinkPHP并非强制使用路由，如果没有定义路由，则可以直接使用“模块/控制器/操作”的方式访问，如果定义了路由，则该路由对应的路由地址就被不能直接访问了。一旦开启强制路由参数，则必须为每个请求定义路由（包括首页）。

使用路由有一定的性能损失，但随之也更加安全，因为每个路由都有自己的生效条件，如果不满足条件的请求是被过滤的。你远比你在控制器的操作中进行各种判断要实用的多。

其实路由的作用远非URL规范这么简单，还可以实现验证、权限、参数绑定及响应设置等功能。

模块

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

支持单一模块架构设计，如果你的应用下面只有一个模块，那么通过配置这个模块的子目录可以省略（同时应用类库的命名空间也随之简化）。

控制器

每个模块拥有独立的类库及配置文件，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，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。

`ThinkPHP`的控制器类比较灵活，可以无需继承任何基础类库。

一个典型的`Index`控制器类如下：

```
namespace app\index\controller;

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

> 继承系统控制器`think\Controller`的话，可以使用内置的功能，享受更多的便利。

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：

```
namespace app\index\controller;

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

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

操作方法可以不使用任何参数，如果定义了一个非可选参数，则该参数必须通过用户请求传入，如果是URL请求，则通常是通过当前的请求传入。

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5.1的架构设计中，只有进行实际的数据库查询操作的时候，才会进行数据库的连接，是真正的惰性连接。

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

视图

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

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

模板引擎

模板文件中可以使用一些特殊的模板标签，这些标签的解析通常由模板引擎负责实现。

ThinkPHP内置了一个基于XML解析的编译型模板引擎，可以很方便的实现模板输出和控制。

同时也可以支持第三方的模板引擎扩展。

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。

> 5.1版本的驱动更多是采用 Composer 的方式安装和管理。

行为

行为（ `Behavior` ）是在预先定义好的一个应用钩子（ `Hook` ）位置执行的一些操作。类似于 `AOP` 编程中的“切面”的概念，给某一个钩子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 `Hook` 位置相关，行为的执行时间取决于行为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\facade\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\facade\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。

事件

ThinkPHP5中的事件一般是指数据库操作和模型操作在完成数据写入之后的回调机制。

数据库操作的回调也称为查询事件，是针对数据库的CURD操作而设计的回调方法，主要包括：

事件	描述
before_select	`select` 查询前回调
before_find	`find` 查询前回调
after_insert	`insert` 操作成功后回调
after_update	`update` 操作成功后回调

after_delete	`delete` 操作成功后回调

模型事件可以看成是模型层的钩子和行为，只不过钩子的位置主要针对模型数据的写入操作，包含下面这些：

钩子	对应操作	快捷注册方法
before_insert	新增前	beforeInsert
after_insert	新增后	afterInsert
before_update	更新前	beforeUpdate
after_update	更新后	afterUpdate
before_write	写入前	beforeWrite
after_write	写入后	afterWrite
before_delete	删除前	beforeDelete
after_delete	删除后	afterDelete

`before_write` 和 `after_write` 表示无论是新增还是更新都会执行的钩子。

助手函数

系统为一些常用的操作提供了助手函数支持，但核心框架本身并不依赖任何助手函数。使用助手函数和性能并无直接影响，只是某些时候无法享受IDE自动提醒的便利，但是否使用助手函数看项目自身规范，在应用的公共函数文件中也可以对系统提供的助手函数进行重写。

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

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

```
// [ 应用入口文件 ]
namespace think;

// 加载基础文件
require __DIR__ . '/../thinkphp/base.php';

// 执行应用并响应
Container::get('app')->run()->send();
```

> 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。

更改应用目录和入口位置

> 新版框架默认不再支持改变应用目录（ `application` ）和入口文件位置，如果你需要更改，需要自己重新定义入口文件。

下面是一个例子（把入口文件放到应用根目录，并且更改应用目录名称为 `app` ）：

```
<?php
namespace think;

// 定义应用目录
define('APP_PATH', __DIR__ . '/app/');
// 加载框架基础引导文件
require __DIR__ . '/thinkphp/base.php';
// 添加额外的代码
// ...

// 执行应用并响应
Container::get('app', [APP_PATH])->run()->send();
```

如果是 `V5.1.2+` 版本，上面的最后一行代码可以使用下面的替代：

```
Container::get('app')->path(APP_PATH)->run()->send();
```

> 更改应用目录名称和位置可能导致默认的命令行操作失效，你需要同步自定义根目录下面的 `think` 文件。

URL访问

URL设计

ThinkPHP `5.1` 在没有定义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。

> 普通模式的URL访问不再支持，但参数可以支持普通方式传值

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

如果不支持PATHINFO的服务器可以使用兼容模式访问如下：

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

必要的时候，我们可以通过某种方式，省略URL里面的模块和控制器。

URL重写

可以通过URL重写隐藏应用的入口文件 `index.php` （也可以是其它的入口文件，但URL重写通常只能设置一个入口文件）,下面是相关服务器的配置参考：

[Apache]

1. `httpd.conf` 配置文件中加载了 `mod_rewrite.so` 模块
2. `AllowOverride None` 将 `None` 改为 `All`
3. 把下面的内容保存为 `.htaccess` 文件放到应用入口文件的同级目录下

```
<IfModule mod_rewrite.c>
```

```
    Options +FollowSymlinks -Multiviews
    RewriteEngine On

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

[IIS]

如果你的服务器环境支持 `ISAPI_Rewrite` 的话，可以配置 `httpd.ini` 文件，添加下面的内容：

```
RewriteRule (.*)$ /index\.php\?s=$1 [I]
```

在IIS的高版本下面可以配置 `web.Config` ，在中间添加 `rewrite` 节点：

```
<rewrite>
 <rules>
 <rule name="OrgPage" stopProcessing="true">
 <match url="^(.*)$" />
 <conditions logicalGrouping="MatchAll">
 <add input="{HTTP_HOST}" pattern="^(.*)$" />
 <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="true" />
 <add input="{REQUEST_FILENAME}" matchType="IsDirectory" negate="true" />
 </conditions>
 <action type="Rewrite" url="index.php/{R:1}" />
 </rule>
 </rules>
 </rewrite>
```

[Nginx]

在Nginx低版本中，是不支持PATHINFO的，但是可以通过在 `Nginx.conf` 中配置转发规则实现：

```
location / { // …..省略部分代码
   if (!-e $request_filename) {
                rewrite  ^(.*)$  /index.php?s=/$1  last;
    }
}
```

> 其实内部是转发到了ThinkPHP提供的兼容URL，利用这种方式，可以解决其他不支持PATHINFO的WEB服务器环境。

如果你的应用安装在二级目录，`Nginx`的伪静态方法设置如下，其中`youdomain`是所在的目录名称。

```
location /youdomain/ {
    if (!-e $request_filename){
        rewrite  ^/youdomain/(.*)$  /youdomain/index.php?s=/$1  last;
    }
}
```

原来的访问URL：

```
http://serverName/index.php/模块/控制器/操作/[参数名/参数值...]
```

设置后，我们可以采用下面的方式访问：

```
http://serverName/模块/控制器/操作/[参数名/参数值...]
```

> 如果你没有修改服务器的权限，可以在`index.php`入口文件做修改，这不是正确的做法，并且不一定成功，视服务器而定，只是在框架执行前补全`$_SERVER['PATH_INFO']`参数。
>
> ```
> $_SERVER['PATH_INFO'] = $_SERVER['REQUEST_URI' ];
> ```

模块设计

> 5.1版本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（可通过环境变量更改）。

目录结构

标准的应用和模块目录结构如下：

```
├─application           应用目录（可设置）
│   ├─common            公共模块目录（可选）
│   ├─module1           模块1目录
│   │   ├─common.php    模块函数文件
│   │   ├─config        模块配置目录（可选）
│   │   ├─controller    控制器目录
│   │   ├─model         模型目录（可选）
│   │   ├─view          视图目录（可选）
│   │   └─ ...          更多类库目录
│   │
│   ├─module2           模块2目录
│   │   ├─common.php    模块函数文件
│   │   ├─config        模块配置目录（可选）
│   │   ├─controller    控制器目录
│   │   ├─model         模型目录（可选）
│   │   ├─view          视图目录（可选）
│   │   └─ ...          更多类库目录
```

> 模块的配置目录也可以放到外面的config目录的模块子目录下面。

遵循ThinkPHP `5.1` 的命名规范，模块目录全部采用小写和下划线命名。

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

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

模块类库

一个模块下面的类库文件的命名空间统一以 `app\模块名` 开头，例如：

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

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，可以在入口文件中进行模块绑定：

```
<?php
// [ 应用入口文件 ]
namespace think;

// 加载基础文件
require __DIR__ . '/../thinkphp/base.php';

// 执行应用并响应（绑定当前访问到index模块）
Container::get('app')->bind('index')->run()->send();
```

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：

```
// 绑定当前访问到index模块的index控制器
Container::get('app')->bind('index/index')->run()->send();
```

设置后，我们的URL访问地址则变成：

> http://serverName/index.php/操作/[参数名/参数值...]

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：

首先在应用配置文件中定义：

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

然后，调整应用目录的结构为如下：

```
├─application           应用目录（可设置）
│   ├─controller        控制器目录
│   ├─model             模型目录
│   ├─view              视图目录
│   ├─ ...              更多类库目录
│   └─common.php        函数文件
```

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。

> 单一模块方式仍然可以通过多级控制器的方式来管理控制器层次

空模块

可以把不存在的模块访问统一指向一个空模块，设置：

```
// 设置空模块名为home
'empty_module'   =>       'home',
```

如果访问了一个不存在的模块，系统会指向home模块进行访问。

> 空模块只有开启多模块访问，并且没有绑定模块的情况下才有效。

环境变量

5.1版本取消了所有的系统常量，原来的系统路径变量改为使用`Env`类获取（需要引入`think\facade\Env`）：

例如：

```
// 获取应用目录（不区分大小写）
echo Env::get('app_path');
// 或者
echo Env::get('APP_PATH');
```

支持获取的系统路径变量包括：

系统路径	Env参数名称
应用根目录	root_path
应用目录	app_path
框架目录	think_path
配置目录	config_path
扩展目录	extend_path
composer目录	vendor_path
运行缓存目录	runtime_path
路由目录	route_path
当前模块目录	module_path

命名空间

命名空间

`ThinkPHP5.1`符合`PSR-4`的自动加载规范，内置不再提供类库文件的导入方法，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);
> ```

从ThinkPHP`5.0`开始，遵循`PSR-4`自动加载规范，只需要给类库正确定义所在的命名空间，并且命名空间的路径与类库文件的目录一致，那么就可以实现类的自动加载，从而实现真正的惰性加载。

例如，`\think\cache\driver\File`类的定义为：

```
namespace think\cache\driver;

class File
{
}
```

如果我们实例化该类的话，应该是：

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

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

> 5.1 版本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。

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

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

```
namespace my;

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

我们就可以直接实例化和调用：

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

如果你的扩展类库不希望放入系统默认的`extend`目录，就需要在应用配置文件中设置`root_namespace`注册根命名空间，例如下面注册了`my`和`org`两个根命名空间到`application/extend`目录下面。

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

应用类库包

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

```
namespace app\index\model;

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

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

```
namespace app\admin\event;

class User
{
}
```

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

如果觉得 `app` 根命名空间不合适或者有冲突，可以更改环境变量 `APP_NAMESPACE` ，如果你定义了 `.env` 文件的话，可以在里面添加：

```
APP_NAMESPACE =  application
```

定义后，应用类库的命名空间改为：

```
<?php
namespace application\index\model;

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

> 本手册后续的章节，均建立在你已经了解PHP命名空间的基础之上，如果不掌握请自行补

> 充PHP基础，对命名空间的缺乏理解会成为你学习ThinkPHP的最大障碍。

容器和依赖注入

容器和依赖注入

`5.1`版本正式引入了容器的概念，用来更方便的管理类依赖及运行依赖注入。

> 5.0版本已经支持依赖注入的，依赖注入和容器没有必然关系

容器类的工作由`think\Container`类完成，但大多数情况我们只需要通过`app`助手函数即可完成大部分操作。

依赖注入其实本质上是指对类的依赖通过构造器完成自动注入，例如在控制器架构方法和操作方法中一旦对参数进行对象类型约束则会自动触发依赖注入，由于访问控制器的参数都来自于URL请求，普通变量就是通过参数绑定自动获取，对象变量则是通过依赖注入生成。

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

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

class Index extends Controller
{
    protected $user;

    public function __construct(User $user)
    {
        $this->user = $user;
    }

    public function hello()
    {
        return 'Hello,' . $this->user->name . '!';
    }
}
```

> 依赖注入的对象参数支持多个，并且和顺序无关。

支持使用依赖注入的场景包括（但不限于）：

- 控制器架构方法；
- 控制器操作方法；

- 数据库和模型事件方法；
- 路由的闭包定义；
- 行为类的方法；

> 在ThinkPHP的设计中， `think\App` 类虽然自身不是容器，但却是一个容器管理类，可以完成容器的所有操作。

绑定

依赖注入的类统一由容器进行管理，你可以随时绑定类到容器中，支持多种绑定方式。

绑定类标识

可以对已有的类库绑定一个标识（唯一），便于快速调用。

```
// 绑定类库标识
bind('cache','think\Cache');
// 快速调用（自动实例化）
$cache = app('cache');
```

> 调用和绑定的标识必须保持一致（包括大小写）

容器中已经调用过的类会自动使用单例，除非你使用下面的方式强制重新实例化。

```
// 每次调用都会重新实例化
$cache = app('cache',true);
```

你可以绑定一个类到容器中（第一个参数直接传入类名）：

```
bind('app\common\Test');
```

但实际上这个操作是多余的，因为只要调用过一次后就会自动绑定

```
app('app\common\Test');
```

> 绑定的类标识可以自己定义（只要不冲突）。

绑定闭包

可以把一个闭包方法绑定到容器中

```
bind('sayHello', function ($name) {
    return 'hello,' . $name;
});
echo app('sayHello',['thinkphp']);
```

绑定类的实例

也可以直接绑定一个类的实例

```
$cache = new think\Cache;
// 绑定类实例
bind('cache',$cache);
// 快速调用类的实例
$cache = app('cache');
```

绑定至接口实现

对于依赖注入使用接口类的情况，我们需要告诉系统使用哪个具体的接口实现类来进行注入，这个使用可以把某个类绑定到接口

```
// 绑定think\LoggerInterface接口实现到think\Log
bind('think\LoggerInterface','think\Log');
```

使用接口作为依赖注入的类型

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

use think\LoggerInterface;

class Index
{
    public function hello(LoggerInterface $log)
    {
        $log->record('hello,world!');
    }
}
```

批量绑定

如果传入一个数组的话，就表示进行批量绑定，例如：

```
bind([
    'route'      => \think\Route::class,
```

```
    'session'    => \think\Session::class,
    'url'        => \think\Url::class,
]);
```

可以在应用或者模块目录下面定义 `provider.php` 文件（返回一个数组），系统会自动批量绑定类库到容器中。

```
return [
    'route'      => \think\Route::class,
    'session'    => \think\Session::class,
    'url'        => \think\Url::class,
];
```

> 绑定标识调用的时候区分大小写，系统已经内置绑定了核心常用类库，无需重复绑定

系统内置绑定到容器中的类库包括

系统类库	容器绑定标识
think\Build	build
think\Cache	cache
think\Config	config
think\Cookie	cookie
think\Debug	debug
think\Env	env
think\Hook	hook
think\Lang	lang
think\Log	log
think\Request	request
think\Response	response
think\Route	route
think\Session	session
think\Url	url
think\Validate	validate
think\View	view

解析

助手函数方式

使用`app`助手函数进行容器中的类解析调用，对于已经绑定的类标识，会自动快速实例化

```
app('cache');
```

上面的app助手函数相当于调用了

```
Container::get('cache');
```

带参数实例化调用

```
app('cache',['file']);
```

对于没有绑定的类，也可以直接解析

```
app('org\utils\ArrayItem');
```

对象化调用

使用`app`助手函数获取容器中的对象实例（支持依赖注入）。

```
$app = app();
// 判断对象实例是否存在
isset($app->cache);

// 注册容器对象实例
$app->cache = think\Cache::class;

// 获取容器中的对象实例
$cache = $app->cache;
```

不带任何参数调用`app`助手函数其实是实例化`think\App`类，可以方便的操作容器、绑定和调用对象实例。

```
// 绑定类到容器
app()->test = new Test;
// 实例调用
$test = app()->test;
```

也就是说，你可以在任何地方使用`app()`方法调用容器中的任何类。

```
// 调用配置类
app()->config->get('app_name');
// 调用session类
app()->session->get('user_name');
```

自动注入

容器的更多使用主要用于依赖注入，和5.0自动注入的方式有所区别，类的绑定操作不再使用`Request`对象而是直接注册到容器中，并且支持模型事件和数据库事件的依赖注入，依赖注入会首先检查容器中是否注册过该对象实例，如果有的话就会自动注入，例如：

我们可以给路由绑定模型对象实例

```
Route::get('user/:id','index/Index/hello')
        ->bindModel('\app\index\model\User');
```

然后在操作方法中自动注入User模型

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

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

class Index extends Controller
{

    public function hello(User $user)
    {
        return 'Hello,'.$user->name;
    }

}
```

Facade

门面（ `Facade` ）

门面为容器中的类提供了一个静态调用接口，相比于传统的静态方法调用， 带来了更好的可测试性和扩展性，你可以为任何的非静态类库定义一个 `facade` 类。

> 系统已经为大部分核心类库定义了 Facade ，所以你可以通过 Facade 来访问这些系统类，当然也可以为你的应用类库添加静态代理。

下面是一个示例，假如我们定义了一个 `app\common\Test` 类，里面有一个 `hello` 动态方法。

```
<?php
namespace app\common;

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

调用hello方法的代码应该类似于：

```
$test = new \app\common\Test;
echo $test->hello('thinkphp'); // 输出 hello, thinkphp
```

接下来，我们给这个类定义一个静态代理类 `app\facade\Test` （这个类名不一定要和 `Test` 类一致，但通常为了便于管理，建议保持名称统一）。

```
<?php
namespace app\facade;

use think\Facade;

class Test extends Facade
{
    protected static function getFacadeClass()
```

```
    {
        return 'app\common\Test';
    }
}
```

只要这个类库继承`think\Facade`，就可以使用静态方式调用动态类`app\common\Test`的动态方法，例如上面的代码就可以改成：

```
// 无需进行实例化 直接以静态方法方式调用hello
echo \app\facade\Test::hello('thinkphp');
```

结果也会输出`hello, thinkphp`。

> 说的直白一点，Facade功能可以让类无需实例化而直接进行静态方式调用。

如果没有通过`getFacadeClass`方法显式指定要静态代理的类，可以在调用的时候进行动态绑定：

```
<?php
namespace app\facade;

use think\Facade;

class Test extends Facade
{
}
```

```
use app\facade\Test;
use think\Facade;

Facade::bind('app\facade\Test', 'app\common\Test');
echo Test::hello('thinkphp');
```

`bind`方法支持批量绑定，因此你可以在应用的公共函数文件中统一进行绑定操作，例如：

```
Facade::bind([
        'app\facade\Test' => 'app\common\Test',
    'app\facade\Info' => 'app\common\Info',
]);
```

核心`Facade`类库

系统给内置的常用类库定义了`Facade`类库，包括：

（动态）类库	Facade类
think\App	think\facade\App
think\Build	think\facade\Build
think\Cache	think\facade\Cache
think\Config	think\facade\Config
think\Cookie	think\facade\Cookie
think\Debug	think\facade\Debug
think\Env	think\facade\Env
think\Hook	think\facade\Hook
think\Lang	think\facade\Lang
think\Log	think\facade\Log
think\Request	think\facade\Request
think\Response	think\facade\Response
think\Route	think\facade\Route
think\Session	think\facade\Session
think\Url	think\facade\Url
think\Validate	think\facade\Validate
think\View	think\facade\View

所以你无需进行实例化就可以很方便的进行方法调用，例如：

```
use think\facade\Cache;

Cache::set('name','value');
echo Cache::get('name');
```

> `think\Db`类的实现本来就类似于`Facade`机制，所以不需要再进行静态代理就可以使用静态方法调用（确切的说`Db`类是没有方法的，都是调用的`Query`类的方法）。

在进行依赖注入的时候，请不要使用`Facade`类作为类型约束，而是建议使用原来的动态类，下面是错误的用法：

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

```
use think\facade\App;

class Index
{
    public function index(App $app)
    {
    }
}
```

应当使用下面的方式：

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

use think\App;

class Index
{
    public function index(App $app)
    {
    }
}
```

为了更加方便的使用系统类库，系统还给这些常用的核心类库的`Facade`类注册了类库别名，当进行静态调用的时候可以直接使用简化的别名进行调用。

别名类	对应Facade类
App	think\facade\App
Build	think\facade\Build
Cache	think\facade\Cache
Config	think\facade\Config
Cookie	think\facade\Cookie
Db	think\Db
Debug	think\facade\Debug
Env	think\facade\Env
Hook	think\facade\Hook
Lang	think\facade\Lang
Log	think\facade\Log
Request	think\facade\Request
Response	think\facade\Response

Route	think\facade\Route
Session	think\facade\Session
Url	think\facade\Url
Validate	think\facade\Validate
View	think\facade\View

因此前面的代码可以改成

```
\Cache::set('name','value');
echo \Cache::get('name');
```

> Facade类定义了一个实例化的 `instance` 方法，如果你的类也有定义的话将会失效。

钩子和行为

钩子和行为

ThinkPHP中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。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，把这些行为抽离出来的目的是为了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钩子，当应用程序运行到这个钩子的时候，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，给某一个钩子绑定相关行为就成了一种类 `AOP` 编程的思想。

一个钩子可以注册多个行为，执行到某个钩子位置后，会按照注册的顺序依次执行相关的行为。但在某些特殊的情况下，你可以设置某个钩子只能执行一次行为，又或者你可以在一个钩子的某个行为中返回 `false` 来强制终止后续的行为执行；一个行为可以同时注册到多个不同的钩子上，完全看应用的需求来设计。

钩子的位置必须是事先设计好的，无论是框架还是应用的，要设置一个钩子，只需要在相关的位置添加一行代码（事先需要引入 `think\facade\Hook` 类）：

```
Hook::listen('钩子名称','参数','是否只有一次有效返回值');
```

除了钩子名称之外，其它参数都是可选的，注意 `5.1` 版本第二个参数不支持引用传值。

系统核心设计提供了一些可能会需要的钩子（位置），尽可能的方便应用的扩展而不必改动框架核心，按照执行顺序依次如下：

钩子	描述	参数
`app_init`	应用初始化标签位	无
`app_dispatch`	应用调度标签位	无
`app_begin`	应用开始标签位	无
`module_init`	模块初始化标签位	无
`action_begin`	控制器开始标签位	当前的callback参数

view_filter	视图输出过滤标签位	当前模板渲染输出内容
app_end	应用结束标签位	当前响应对象实例
log_write	日志write方法标签位	当前写入的日志信息
log_write_done	日志写入完成标签位	
response_send	响应发送标签位	当前响应对象
response_end	输出结束标签位	当前响应对象实例

> 其中 log_write 钩子仅在调用 Log::write 方法的时候执行。
>
> > view_filter 钩子 v5.1.3+ 版本中已经废除，改用视图类的 filter 方法过滤。

行为定义

行为类的定义很简单，一般来说只需要定义一个行为入口方法 run 即可，例如：

```
namespace app\index\behavior;

class Test
{
    public function run($params)
    {
        // 行为逻辑
    }
}
```

可以在行为方法中使用依赖注入，例如：

```
namespace app\index\behavior;

use think\Request;

class Test
{
    public function run(Request $request, $params)
    {
        // 行为逻辑
    }
}
```

行为的入口方法名称支持自定义，如果需要更改在应用公共文件中添加下面的代码即可：

```
Hook::portal('portal');
```

入口方法名称就变成了`portal`。

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钩子，可以采用如下定义：

```
namespace app\index\behavior;

class Test
{
    public function appInit($params)
    {

    }

    public function appEnd($params)
    {

    }
}
```

该行为绑定到`app_init`和`app_end`钩子后就会调用相关的方法，方法名就是钩子名称的驼峰命名（首字母小写）。

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`think\facade\Hook`类的add方法注册行为，例如：

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

如果要批量注册行为的话，可以使用：

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

当应用运行到`app_init`标签位的时候，就会依次调用`app\index\behavior\CheckAuth`、`app\index\behavior\CheckLang`和`app\admin\behavior\CronRun`行为。如果其中一个行为中有中止代码的话则后续不会执行，如果返回`false`则当前标签位的后续行为将不会执行，但应用将继续运行。

我们也可以直接在应用目录下面或者模块的目录下面定义`tags.php`文件来统一定义行为，定义格式如下：

```
return [
    'app_init'=> [
        'app\\index\\behavior\\CheckAuth',
        'app\\index\\behavior\\CheckLang'
    ],

    'app_end'=> [
        'app\\admin\\behavior\\CronRun'
    ]
]
```

如果应用目录下面和模块目录下面的`tags.php`都定义了`app_init`的行为绑定的话，会采用合并模式，如果希望覆盖，那么可以在模块目录下面的`tags.php`中定义如下：

```
return [
    'app_init'=> [
        'app\\index\\behavior\\CheckAuth',
        '_overlay'=>true
    ],
    'app_end'=> [
        'app\\admin\\behavior\\CronRun'
    ]
]
```

如果某个行为标签定义了`'_overlay' =>true`就表示覆盖之前的相同标签下面的行为定义。

闭包支持

可以不用定义行为直接把闭包函数绑定到某个标签位，例如：

```
Hook::add('app_init',function(){
    echo 'Hello,world!';
});
```

如果标签位有传入参数的话，闭包也可以支持传入参数，例如：

```
Hook::listen('action_init',$params);
Hook::add('action_init',function($params){
    var_dump($params);
});
```

直接执行行为

如果需要，你也可以不绑定行为标签，直接调用某个行为，使用：

```
// 执行 app\index\behavior\CheckAuth行为类的run方法 并引用传入params参数
$result = Hook::exec('app\\index\\behavior\\CheckAuth',$params);
```

直接执行行为的时候，执行的是run方法，如果需要执行行为类的其它方法，可以使用

```
// 执行 app\index\behavior\CheckAuth行为类的hello方法 并引用传入params参数
$result = Hook::exec(['app\\index\\behavior\\CheckAuth','hello'], $params
);
```

路由

路由是应用开发中比较关键的一个环节，其主要作用包括但不限于：

- 让URL更规范以及优雅；
- 隐式传入额外请求参数；
- 统一拦截并进行权限检查等操作；
- 绑定请求数据；
- 使用请求缓存；

路由解析的过程一般包含：

> - 路由定义：完成路由规则的定义和参数设置（5.1的路由定义采用了对象化的思维，相对5.0而言更直观）；
> - 路由检测：检查当前的URL请求是否有匹配的路由；
> - 路由解析：解析当前路由实际对应的操作（方法或闭包）；
> - 路由调度：执行路由解析的结果调度（主业务逻辑）；

掌握路由主要是要掌握路由定义及参数设置，其它环节是由系统自动完成的。

> 路由的主体规划和定义应该尽可能在应用开发前完成，在后期可以进行路由的参数调整和规则增补。

路由仅针对PATHINFO方式的URL有效，`ThinkPHP5.1`的路由定义更加对象化，并且默认开启路由（不能关闭），如果一个URL没有定义路由，则采用默认的`PATH_INFO`模式访问URL：

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

> 在不使用路由的情况下，仍然可以通过操作方法的参数绑定、空控制器和空操作等特性实现URL地址的简化（参考后面的请求->参数绑定章节）。

强制路由

在`app.php`配置文件中设置

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

将开启强制使用路由，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!';
});
```

延迟解析

如果你定义了太多的路由，担心影响性能，可以开启路由的延迟解析功能，只需要在`app.php`配置文件中设置：

```
// 开启路由延迟解析
'url_lazy_route'        => true,
```

> 尽量通过路由分组或者域名域名来定义路由才能发挥延迟解析的优势。

一旦开启路由的延迟解析，将会对定义的域名路由和分组路由进行延迟解析，也就是说只有实际匹配到该域名或者分组后才会进行路由规则的注册，避免不必要的注册和解析开销。

> 推荐的方式是开发模式下关闭延迟解析，部署后开启并生成路由映射缓存。

> 开启路由延迟解析后，将会导致你的URL生成无法准确识别路由规则的反解，但可以通过路由映射缓存指令（参考命令行章节的生成路由映射缓存一节）来解决。

路由定义

注册路由规则

> route 目录下的任何路由定义文件都是有效的，默认的路由定义文件是 route.php ，但你完全可以更改文件名，或者添加多个路由定义文件（你可以进行模块定义区分，但最终都会一起加载）。

```
├─route                    路由定义目录
│   ├─route.php            路由定义
│   ├─api.php              路由定义
│   └─...                  更多路由定义
```

假设后面的路由定义内容我们统一在 route.php 文件里面定义，最基础的路由定义方法是：

> Route::rule('路由表达式','路由地址','请求类型');

除了路由表达式和路由地址是必须的外，其它参数均为可选。 5.1 版本推荐采用方法的方式定义请求类型、路由参数及变量规则。

例如注册如下路由规则：

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

我们访问：

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

会自动路由到：

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

并且原来的访问地址会自动失效。

可以在 `rule` 方法中指定请求类型（不指定的话默认为任何请求类型有效），例如：

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

> 请求类型参数不区分大小写。

表示定义的路由规则在 `POST` 请求下才有效。

如果要定义 `GET` 和 `POST` 请求支持的路由规则，可以用：

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

不过通常我们更愿意使用对应请求类型的快捷方法，包括：

类型	描述	快捷方法
GET	GET请求	get
POST	POST请求	post
PUT	PUT请求	put
DELETE	DELETE请求	delete
PATCH	PATCH请求	patch
*	任何请求类型	any

快捷注册方法的用法为：

> Route::快捷方法名('路由表达式','路由地址');

使用示例如下：

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

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执行最终的调度方法，后续规则就不再检测。

路由表达式

路由表达式统一使字符串定义，采用规则定义的方式（不支持直接使用正则表达式，但支持给某个变量定义正则，参考后面的变量规则部分）。

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：

```
Route::rule('/', 'index'); // 首页访问路由
Route::rule('my', 'Member/myinfo'); // 静态地址路由
Route::rule('blog/:id', 'Blog/read'); // 静态地址和动态地址结合
Route::rule('new/:year/:month/:day', 'News/read'); // 静态地址和动态地址结合
Route::rule(':user/:blog_id', 'Blog/read'); // 全动态地址
```

> 规则表达式的定义以 / 为参数分割符（无论你的 PATH_INFO 分隔符设置是什么，请确保在定义路由规则表达式的时候统一使用 / 进行URL参数分割，除非是使用组合变量的情况）。

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

> 你的URL访问 PATH_INFO 分隔符使用 pathinfo_depr 配置，但无论如何配置，都不影响路由的规则表达式的路由分隔符定义。

可选定义

支持对路由参数的可选定义，例如：

```
Route::get('blog/:year/[:month]','Blog/archive');
```

变量用`[ ]`包含起来后就表示该变量是路由匹配的可选变量。

以上定义路由规则后，下面的URL访问地址都可以被正确的路由匹配：

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

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

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

完全匹配

规则匹配检测的时候默认只是对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URL进行完全匹配，可以在路由表达式最后使用`$`符号，例如：

```
Route::get('new/:cate$', 'News/category');
```

这样定义后

```
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

```
Route::get('new/:cate', 'News/category');
```

方式定义的话，则两种方式的URL访问都可以匹配成功。

如果需要全局进行URL完全匹配，可以在`app.php`中设置

```
    // 开启路由完全匹配
    'route_complete_match'   => true,
```

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：

```
Route::get('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。

路由标识

如果你需要快速的根据路由生成URL地址，可以在定义路由的时候指定生成标识（但要确保唯一）。

例如

```
// 注册路由到index模块的News控制器的read操作
Route::name('new_read')->rule('new/:id','index/News/read');
```

生成路由地址的时候就可以使用

```
url('new_read',['id'=>10]);
```

如果不定义路由标识的话，使用下面的方式生成

```
url('index/News/read',['id'=>10]);
```

变量规则

变量规则

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

局部变量规则

局部变量规则，仅在当前路由有效：

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

> 不需要开头添加 ^ 或者在最后添加 $ ，系统会自动添加，但也可以使用完整正则定义。

如果需要定义一个完整的路由正则，例如需要指定模式修饰符，可以使用下面的方式：

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

> 以 / 开头的正则规则表示使用完整路由规则，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。

全局变量规则

设置全局变量规则，全部路由有效：

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

组合变量规则

如果你的路由规则比较特殊，可以在路由定义的时候使用组合变量。

例如：

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

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

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

使用组合变量的情况下如果需要使用可选变量，则可以使用下面的方式：

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

> 组合变量的正则定义不支持使用模式修饰符

动态路由

可以把路由规则中的变量传入路由地址中，就可以实现一个动态路由，例如：

```
// 定义动态路由
Route::get('hello/:name', 'index/:name/hello');
```

`name` 变量的值作为路由地址传入。

动态路由中的变量也支持组合变量及拼装，例如：

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

路由地址

路由地址

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

定义方式	定义格式
方式1：路由到模块/控制器	'[模块/控制器/操作]?额外参数1=值1&额外参数2=值2...'
方式2：路由到重定向地址	'外部地址'（默认301重定向）或者 ['外部地址','重定向代码']
方式3：路由到控制器的方法	'@[模块/控制器/]操作'
方式4：路由到类的方法	'\完整的命名空间类::静态方法' 或者 '\完整的命名空间类@动态方法'
方式5：路由到闭包函数	闭包函数定义（支持参数传入）
方式6：路由到Response对象	Response对象定义及设置
方式7：路由到模板文件	使用 `view` 方法（ `V5.1.3+` ）

其中方式5和6我们将会在下一节闭包支持中详细描述。

路由到模块/控制器/操作

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

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

路由地址的格式为：

> [模块/控制器/]操作?参数1=值1&参数2=值2...

解析规则是从操作开始解析，然后解析控制器，最后解析模块，例如：

```
// 路由到默认或者绑定模块
Route::get('blog/:id','blog/read');
// 路由到index模块
Route::get('blog/:id','index/blog/read');
```

Blog类定义如下：

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

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

路由地址中支持多级控制器，使用下面的方式进行设置：

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

表示路由到下面的控制器类，

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

Blog类定义如下：

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

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

还可以支持路由到动态的模块、控制器或者操作，例如：

```
// action变量的值作为操作方法传入
Route::get(':action/blog/:id', 'index/blog/:action');
// 变量传入index模块的控制器和操作方法
Route::get(':c/:a', 'index/:c/:a');
```

额外参数

在这种方式路由跳转的时候支持额外传入参数对（额外参数指的是不在URL里面的参数，隐

式传入需要的操作中，有时候能够起到一定的安全防护作用，后面我们会提到）。例如：

```
Route::get('blog/:id','blog/read?status=1&app_id=5');
```

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

路由到操作方法

路由地址的格式为：

> @[模块/控制器/]操作

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

例如，定义如下路由后：

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

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

Blog类定义如下：

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

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

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

路由到类的方法

路由地址的格式为（动态方法）：

> \类的命名空间\类名@方法名

或者（静态方法）

> \类的命名空间\类名::方法名

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

```
Route::get('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：

```
Route::get('blog/:id','\app\index\service\Blog::read');
```

支持传入额外的参数作为方法的参数调用（用于参数绑定），例如：

```
Route::get('blog/:id','\app\index\service\Blog::read?status=1');
```

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：

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

和

```
Route::get('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：

```
Route::get('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：

```
Route::get('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` 。

路由重定向默认使用301状态吗，可以使用status方法单独设置，例如：

```
Route::get('blog/:id','http://blog.thinkphp.cn/read/:id')->status(302);
```

`V5.1.3+` 版本开始，可以直接使用 `redirect` 方法注册一个重定向路由

```
Route::redirect('blog/:id','http://blog.thinkphp.cn/read/:id',302);
```

路由到模板（ `V5.1.3` ）

`V5.1.3+` 版本开始，支持路由直接渲染模板输出。

```
Route::view('hello/:name','index@hello');
```

表示该路由会渲染 index模块下面的 `view/hello.html` 模板文件输出。

模板文件中可以直接输出当前请求的 `param` 变量，如果需要增加额外的模板变量，可以使用：

```
Route::view('hello/:name','index@hello',['city'=>'shanghai']);
```

在模板中可以输出 `name` 和 `city` 两个变量。

```
Hello,{$name}--{$city}!
```

闭包支持

闭包定义

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

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

参数传递

闭包定义的时候支持参数传递，例如：

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

规则路由中定义的动态变量的名称 就是闭包函数中的参数名称，不分次序。

因此，如果我们访问的URL地址是：

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

则浏览器输出的结果是：

```
Hello,thinkphp
```

依赖注入

可以在闭包中使用依赖注入，例如：

```
Route::rule('hello/:name', function (Request $request, $name) {
    $method = $request->method();
    return '[' . $method . '] Hello,' . $name;
});
```

指定响应对象

更多的情况是在路由闭包中指定响应对象输出，例如：

```
Route::get('hello/:name', function (Response $response, $name) {
    return $response
        ->data('Hello,' . $name)
        ->code(200)
        ->contentType('text/plain');
});
```

这种情况可以直接写成

```
Route::get('hello/:name', response()
        ->data('Hello,' . $name)
        ->code(200)
        ->contentType('text/plain'));
```

更多的情况是直接对资源文件的请求设置404访问

```
// 对于不存在的static目录下的资源文件设置404访问
Route::get('static', response()->code(404));
```

路由参数

路由参数

路由分组及规则定义支持指定路由参数，这些参数主要完成路由匹配检测以及行为执行。

`5.1` 版本极大改进了路由参数的用法。

> 路由参数可以在定义路由规则的时候直接传入（批量），不过 `5.1` 采用了更加面向对象的方式进行路由参数配置，因此使用方法配置更加清晰。

参数	说明	方法名
method	请求类型检测，支持多个请求类型	method
ext	URL后缀检测，支持匹配多个后缀	ext
deny_ext	URL禁止后缀检测，支持匹配多个后缀	denyExt
https	检测是否https请求	https
domain	域名检测	domain
before	前置行为（检测）	before
after	后置行为（执行）	after
merge_extra_vars	合并额外参数	mergeExtraVars
complete_match	是否完整匹配路由	completeMatch
model	绑定模型	model
cache	请求缓存	cache
param_depr	路由参数分隔符	depr
ajax	Ajax检测	ajax
pjax	Pjax检测	pjax
response	绑定response_send行为	response
validate	绑定验证器类进行数据验证	validate
header	设置Response的header信息	header
append	追加额外的参数（`5.1.5+`）	append

> `ext` 和 `deny_ext` 参数允许设置为空，分别表示不允许任何后缀以及必须使用后缀访问。

用法举例：

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

和使用方法设置（新版推荐的设置方式）等效

```
Route::get('new/:id', 'News/read')
    ->ext('html')
    ->https();
```

显然第二种方式更加直观，而且便于IDE的自动提示。

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

URL后缀

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

支持匹配多个后缀，例如：

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

> 如果`ext`方法不传入任何值，表示不允许使用任何后缀访问。

可以设置禁止访问的URL后缀，例如：

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

> 如果`denyExt`方法不传入任何值，表示必须使用后缀访问。

域名检测

支持使用完整域名或者子域名进行检测，例如：

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

> 如果需要给子域名定义批量的路由规则，建议使用`domain`方法进行路由定义。

HTTPS检测

支持检测当前是否HTTPS访问

```
// 必须使用HTTPS访问
Route::get('new/:id', 'News/read')
    ->https();

// 必须使用HTTP访问
Route::get('new/:id', 'News/read')
    ->https(false);
```

前置行为检测

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

```
Route::get('user/:id', 'index/User/read')
    ->before(['\app\index\behavior\UserCheck']);
```

行为类定义如下：

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

class UserCheck
{
    public function run()
    {
        if ('user/0' == request()->url()) {
            return false;
        }
    }
}
```

可以同时使用多个行为进行检测，并且支持使用闭包。

> 因为前置行为的特殊性，在路由参数的有效性检查后，无论是否最终匹配该路由，都会进行前置行为检查（路由分组的话 会在匹配改分组后 再检查）。

后置行为执行

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

```
Route::get('user/:id', 'User/read')
    ->after(['\app\index\behavior\ReadInfo']);
```

其中`ReadInfo`行为类定义如下：

```
<?php

namespace app\index\behavior;

use app\index\model\User;

class ReadInfo
{
    public function run()
    {
        $id             = request()->route('id');
        app()->user = User::get($id);
    }
}
```

如果成功匹配到`new/:id`路由后，就会执行行为类的`run`方法，参数是路由地址，可以动态改变。同样，后置行为也支持传入闭包。

合并额外参数

通常用于完整匹配的情况，如果有额外的参数则合并作为变量值，例如：

```
Route::get('new/:name$', 'News/read')
    ->mergeExtraVars();
```

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

会被匹配到，并且`name`变量的值为`thinkphp/hello`。

路由绑定模型

路由规则和分组支持绑定模型数据，例如：

```
Route::get('hello/:id', 'index/index/hello')
    ->model('id', '\app\index\model\User');
```

会自动给当前路由绑定 `id` 为 当前路由变量值的 `User` 模型数据。

默认情况下，如果没有查询到模型数据，则会抛出异常，如果不希望抛出异常，可以使用

```
Route::rule('hello/:id', 'index/index/hello')
    ->model('id', '\app\index\model\User', false);
```

可以定义模型数据的查询条件，例如：

```
Route::rule('hello/:name/:id', 'index/index/hello')
    ->model('id&name', '\app\index\model\User');
```

表示查询 `id` 和 `name` 的值等于当前路由变量的模型数据。

也可以使用闭包来自定义返回需要的模型对象

```
Route::rule('hello/:id', 'index/index/hello')
    ->model(function ($param) {
        $model = new \app\index\model\User;
        return $model->where($param)->find();
    });
```

闭包函数的参数就是当前请求的URL变量信息。

> 绑定的模型可以直接在控制器的架构方法或者操作方法中自动注入，具体可以参考请求章节的依赖注入。

缓存路由请求

可以对当前的路由请求进行缓存处理，例如：

```
Route::get('new/:name$', 'News/read')
    ->cache(3600);
```

表示对当前路由请求缓存3600秒，更多内容可以参考请求缓存一节。

设置Header信息

```
Route::get('new/:name$', 'News/read')
    ->header('Access-Control-Allow-Origin','*');
```

> header方法支持多次调用。

或者使用数组方式批量设置

```
Route::get('new/:name$', 'News/read')
    ->header([
        'Access-Control-Allow-Origin'=>'*',
        'Access-Control-Allow-Methods' => 'GET, POST, PATCH, PUT, DELETE'
,
    ]);
```

当路由匹配后，会自动设置本次请求的Response响应对象的Header信息。

响应输出设置

可以调用 `response` 方法给路由或者分组绑定响应输出参数，例如：

```
Route::get('hello/:id', 'index/index/hello')->response([
    '\app\index\behavior\Json',
]);
```

行为类定义如下：

```
namespace app\index\behavior;

class Json
{
    public function run($response)
    {
        // 调用Response类的方法进行设置
        $response->contentType('application/json');
    }
}
```

如果不希望使用行为，可以直接使用闭包

```
Route::get('hello/:id', 'index/index/hello')->response(function($response
){
        $response->contentType('application/json');
});
```

如果要给某个路由返回单独的响应对象，也可以使用

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

全局路由参数

可以直接进行全局的路由参数设置，例如在路由定义文件中使用

```
Route::option('ext','html')->option('cache', 600);
```

表示全局路由都使用html后缀以及600秒的请求缓存。

动态参数

如果你需要额外自定义一些路由参数，可以使用下面的方式：

```
Route::get('new/:name$', 'News/read')
    ->option('rule','admin');
```

或者使用动态方法

```
Route::get('new/:name$', 'News/read')
    ->rule('admin');
```

在后续的路由行为后可以调用该路由的`rule`参数来进行权限检查。

跨域请求

跨域请求

如果某个路由或者分组需要支持跨域请求，可以使用

```
Route::get('new/:id', 'News/read')
    ->ext('html')
    ->allowCrossDomain();
```

> 跨域请求一般会发送一条 OPTIONS 的请求，一旦设置了跨域请求的话，不需要自己定义 OPTIONS 请求的路由，系统会自动加上。

跨域请求系统会默认带上一些Header，包括：

```
Access-Control-Allow-Origin:*
Access-Control-Allow-Methods:GET, POST, PATCH, PUT, DELETE
Access-Control-Allow-Headers:Authorization, Content-Type, If-Match, If-Mo
dified-Since, If-None-Match, If-Unmodified-Since, X-Requested-With
```

你可以添加或者更改Header信息，使用

```
Route::get('new/:id', 'News/read')
    ->ext('html')
    ->header('Access-Control-Allow-Origin','thinkphp.cn')
    ->header('Access-Control-Allow-Credentials', 'true')
    ->allowCrossDomain();
```

注解路由

注解路由

新版本提供了一种最简单的路由注册方法（可以完成基本的路由定义），默认关闭，如果需要开启在应用的`app.php`配置文件中设置：

```
    // 使用注解路由
    'route_annotation'       => true,
```

然后只需要直接在控制器类的方法注释中定义（通常称之为注解路由），例如：

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

class Index
{
    /**
     * @param  string  $name 数据名称
     * @return mixed
     * @route('hello/:name')
     */
        public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

请务必注意注释的规范，可以利用IDE生成规范的注释。

> 该方式定义的路由在调试模式下面实时生效，部署模式则需要使用 `optimize::route` 指令生成路由规则文件。

注意必须严格使用`@route(`（区分大小写，`route`和`(`之间不能有空格），建议路由定义写在注释最后一段，否则后面需要一个空行。

然后就使用下面的URL地址访问：

```
http://tp5.com/hello/thinkphp
```

页面输出

```
hello,thinkphp
```

默认注册的路由规则是支持所有的请求，如果需要指定请求类型，可以在第二个参数中指定请求类型：

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

class Index
{
    /**
     * @param  string  $name 数据名称
     * @return mixed
     * @route('hello/:name','get')
     */
        public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

如果有路由参数和变量规则需要定义，可以直接在后面添加方法，例如：

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

class Index
{
    /**
     * @param string        $name 数据名称
     * @route('hello/:name','get')
     *  ->https()
     *  ->pattern(['name' => '\w+'])
     *
     * @return mixed
     */
        public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

> 注意在添加路由参数和变量规则的最后不需要加`;`，并且确保和后面的其它注释之间间隔一个空行。

支持在类的注释里面定义资源路由，例如：

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

/**
 * @route('blog')
 */
class Blog
{
    public function index()
    {
    }

    public function read($id)
    {
    }

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

路由分组

路由分组

路由分组功能允许把相同前缀的路由定义合并分组，这样可以简化路由定义，并且提高路由匹配的效率，不必每次都去遍历完整的路由规则（尤其是开启了路由延迟解析后性能更佳）。

使用`Route`类的`group`方法进行注册，给分组路由定义一些公用的路由设置参数，使用规范如下：

> Route::group('分组名（字符串）或者分组路由参数（数组）','分组路由规则（数组或者闭包）');

其中路由参数和变量规则为可选，同样建议使用方法进行参数设置。

例如：

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

支持使用闭包方式注册路由分组，例如：

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

如果仅仅是用于对一些路由规则设置一些公共的路由参数，也可以使用：

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

路由分组支持嵌套，例如：

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

> 如果使用了嵌套分组的情况，子分组会继承父分组的参数和变量规则，而最终的路由规则里面定义的参数和变量规则为最优先。

延迟路由解析

新版路由支持延迟路由解析，也就是说你定义的路由规则（主要是分组路由和域名路由规则）在加载路由定义文件的时候并没有实际注册，而是在匹配到路由分组或者域名的情况下，才会实际进行注册和解析，大大提高了路由注册和解析的性能。

默认是关闭延迟路由解析的，你可以在应用的`app.php`配置文件中设置：

```
// 开启路由延迟解析
'url_lazy_route'         => true,
```

开启延迟路由解析后，如果你需要生成路由反解URL，需要使用命令行指令

```
php think optimize:route
```

来生成路由缓存解析。

传入额外参数

可以统一给分组路由传入额外的参数

```
Route::group('blog', [
    ':id'   => 'Blog/read',
    ':name' => 'Blog/read',
])->ext('html')
->pattern(['id' => '\d+'])
->append(['group_id'=>1]);
```

上面的分组路由统一传入了`group_id`参数，该参数的值可以通过`Request`类的`param`方法获取。

MISS路由

全局MISS路由

如果希望在没有匹配到所有的路由规则后执行一条设定的路由，可以注册一个单独的`MISS`路由：

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

> 一旦设置了MISS路由，相当于开启了强制路由模式

当所有的路由规则都没有匹配到后，会路由到`public/miss`路由地址。

> 全局路由其实是针对域名的，只不过默认是针对当前访问的域名，可以在域名路由中单独设置MISS路由

分组MISS路由

分组支持独立的`MISS`路由，例如如下定义：

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

资源路由

资源路由

支持设置`RESTFul`请求的资源路由，方式如下：

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

表示注册了一个名称为`blog`的资源路由到`index`模块的`Blog`控制器，系统会自动注册7个路由规则，如下：

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

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

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

Blog控制器中的对应方法如下：

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

class Blog
{
    public function index()
    {
    }

    public function read($id)
    {
```

```
    }

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

可以通过命令行快速创建一个资源控制器类（参考后面的控制器章节的资源控制器一节）。

可以改变默认的id参数名，例如：

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

控制器的方法定义需要调整如下：

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

class Blog
{
    public function index()
    {
    }

    public function read($blog_id)
    {
    }

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

也可以在定义资源路由的时候限定执行的方法（标识），例如：

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

// 排除index和delete操作
Route::resource('blog', 'index/blog')
    ->except(['index', 'delete']);
```

资源路由的标识不可更改，但生成的路由规则和对应操作方法可以修改。

如果需要更改某个资源路由标识的对应操作，可以使用下面方法：

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

设置之后，URL访问变为：

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

创建blog页面的对应的操作方法也变成了add。

支持批量更改，如下：

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

资源嵌套

支持资源路由的嵌套，例如：

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

就可以访问如下地址：

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

生成的路由规则分别是：

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

Comment控制器对应的操作方法如下：

```
<?php
```

```
namespace app\index\controller;

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

edit方法中的参数顺序可以随意，但参数名称必须满足定义要求。

如果需要改变其中的变量名，可以使用：

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

Comment控制器对应的操作方法改变为：

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

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

快捷路由

快捷路由

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

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

User控制器定义如下：

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

class User
{
    public function getInfo()
    {
    }

    public function getPhone()
    {
    }

    public function postInfo()
    {
    }

    public function putInfo()
    {
    }

    public function deleteInfo()
    {
    }
}
```

我们可以通过下面的URL访问

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

路由别名

路由别名

路由别名功能可以使用一条规则，批量定义一系列的路由规则。

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

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

然后可以直接通过URL地址访问User控制器的操作，例如：

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

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

路由别名可以指向任意一个有效的路由地址，例如下面指向一个类

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

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

支持给路由别名设置路由条件，例如：

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

操作方法黑白名单

路由别名的操作方法支持白名单或者黑名单机制，例如：

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

或者使用黑名单机制

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

并且支持设置操作方法的请求类型，例如：

```
// user 别名路由到 index/user 控制器
Route::alias('user', 'index/user', [
    'ext'    => 'html',
    'allow'  => 'index,save,delete',
    'method' => ['index' => 'GET', 'save' => 'POST', 'delete' => 'DELETE'
],
]);
```

路由绑定

可以使用路由绑定简化URL或者路由规则的定义，绑定支持如下方式：

绑定到模块/控制器/操作

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

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

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

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

可以简化成

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

如果定义了路由

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

那么访问URL就变成了

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

绑定到命名空间

把当前的URL绑定到某个指定的命名空间，例如：

```
// 绑定命名空间
```

```
Route::bind(':\app\index\controller');
```

那么，我们接下来只需要通过

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

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

绑定到类

把当前的URL直接绑定到某个指定的类，例如：

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

那么，我们接下来只需要通过

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

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

> 注意：绑定到命名空间和类之后，不会进行模块的初始化工作。

入口文件绑定

如果我们需要给某个入口文件绑定模块，可以使用下面两种方式：

手动添加绑定

把当前入口文件绑定到指定的模块或者控制器，例如：

```
// [ 应用入口文件 ]
namespace think;

// 加载基础文件
require __DIR__ . '/../thinkphp/base.php';

// 支持事先使用静态方法设置Request对象和Config对象

// 绑定到index模块 执行应用并响应
Container::get('app')->bind('index')->run()->send();
```

也支持绑定到控制器

```
Container::get('app')->bind('index/index')->run()->send();
```

自动入口绑定

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

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

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

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

域名路由

域名路由

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

可以单独给域名设置路由规则，例如给blog子域名注册单独的路由规则：

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

> 闭包中可以使用路由的其它方法，包括路由分组，甚至给域名设置MISS路由。

支持数组的方式：

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

支持同时对多个域名设置相同的路由规则：

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

如果你需要设置一个路由跨所有域名都可以生效，可以对分组路由或者某个路由使用 `crossDomainRule` 方法设置：

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

域名绑定

绑定到模块

除了设置域名的路由规则之外，还支持单独给域名进行路由绑定，例如绑定到模块：

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

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

// IP绑定到admin模块
Route::domain('114.23.4.5', 'admin');
```

blog子域名绑定后，URL访问规则变成：

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

支持直接绑定到控制器，例如：

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

URL访问地址变化为：

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

如果你的域名后缀比较特殊，例如是 `com.cn` 或者 `net.cn` 之类的域名，需要在应用配置文件 `app.php` 中配置：

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

同样也支持绑定到命名空间或者类

绑定到命名空间

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

绑定到类

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

泛域名部署

可以支持泛域名部署规则，例如：

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

下面的URL访问都会直接访问book模块

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

并且可以直接通过 `Request::panDomain()` 获取当前的泛域名值。

支持三级泛域名部署，例如：

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

也支持直接把泛域名的值作为额外参数传入

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

就可以通过 `Request::param('name')` 获取当前泛域名的值

> 目前只支持二级域名和三级域名的泛域名部署。

绑定到Response对象

可以直接绑定某个域名到 `Response` 对象，例如：

```
// 绑定域名到Response对象
Route::domain('test', response()->code(404));
```

如果域名需要同时定义路由规则，并且对其它的情况进行绑定操作，可以在闭包里面执行绑定操作，例如：

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

在 `blog` 域名下面定义了一个 `new/:id` 的路由规则，指向 `index` 模块，而其它的路由则绑定到 `blog` 模块。

传入额外参数

可以在域名绑定或者路由定义后传入额外的隐藏参数，例如：

```
Route::domain('blog', function () {
    // 动态注册域名的路由规则
    Route::rule('new/:id', 'index/news/read');
    Route::rule(':user', 'index/user/info');
})->append(['app_id'=>1]);
```

上面的域名路由统一传入了 `app_id` 参数，该参数的值可以通过 `Request` 类的 `param` 方法获取。

也可以直接在域名绑定后传入额外参数

```
Route::domain('blog', 'blog')
        ->append(['app_id'=>1]);
```

URL生成

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

> 如果你开启了路由延迟解析，需要生成路由映射缓存才能支持全部的路由地址的反转解析。

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

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

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

地址表达式和参数

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

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

就可以使用下面的方式来生成URL地址：

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

下面我们统一使用第一种方式讲解。

使用模块/控制器/操作生成

如果你的路由方式是路由到模块/控制器/操作，那么可以直接写

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

以上方法都会生成下面的URL地址：

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

> 注意，生成方法的第一个参数必须和路由定义的路由地址保持一致，如果写成下面的方式可能无法正确生成URL地址：
>
> ```
> Url::build('blog/read','id=5&name=thinkphp');
> ```

如果你的环境支持REWRITE，那么生成的URL地址会变为：

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

如果你配置了：

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

那么生成的URL地址变为：

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

不在路由规则里面的变量会直接使用普通URL参数的方式。

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

使用控制器的方法生成

如果你的路由地址是采用控制器的方法，并且路由定义如下：

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

那么可以使用如下方式生成：

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

那么自动生成的URL地址变为：

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

使用类的方法生成

如果你的路由地址是路由到类的方法，并且做了如下路由规则定义：

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

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

那么可以使用如下方式生成：

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

那么自动生成的URL地址变为：

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

直接使用路由地址

我们也可以直接使用路由地址来生成URL，例如：

我们定义了路由规则如下：

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

可以使用下面的方式直接使用路由规则生成URL地址：

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

那么自动生成的URL地址变为：

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

URL后缀

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

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

那么自动生成的URL地址变为：

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

如果我们设置了多个URL后缀支持

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

则会取第一个后缀来生成URL地址，所以自动生成的URL地址还是：

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

如果你希望指定URL后缀生成，则可以使用：

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

域名生成

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

自动生成域名

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

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

例如，我们注册了域名路由信息如下：

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

那么上面的URL地址生成为：

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

指定域名

你也可以显式传入需要生成地址的域名，例如：

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

或者传入完整的域名

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

生成的URL地址为：

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

也可以直接在第一个参数里面传入域名，例如：

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

生成锚点

支持生成URL的锚点，可以直接在URL地址参数中使用：

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

> 锚点和域名一起使用的时候，注意锚点在前面，域名在后面。

生成的URL地址为：

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

隐藏或者加上入口文件

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

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

或者隐藏`index.php`：

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

> `root`方法只需要调用一次即可。

控制器

按照ThinkPHP的架构设计，所有的URL请求（无论是否采用了路由），最终都会定位到控制器（也许实际的类不一定是控制器类，但也属于广义范畴的控制器）。控制器的层可能有很多，为了便于区分就把通过URL访问的控制器称之为访问控制器（通常意义上我们所说的控制器就是指访问控制器）。

ThinkPHP `V5.1` 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，或者根据业务需求封装自己的基础控制器类。

> 本章的大部分内容是以基础系统的控制器基类为前提，但很多情况并非必须，系统也提供了很多助手函数帮助你在不继承系统控制器基类的情况下实现相同的功能。

控制器定义

控制器定义

控制器文件通常放在`application/module/controller`下面，类名和文件名保持大小写一致，并采用驼峰命名（首字母大写）。

一个典型的控制器类定义如下：

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

use think\Controller;

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

> 为了更方便使用，控制器类建议继承系统的控制器基类`think\Controller`，虽然无需继承也可以使用。

控制器类文件的实际位置是

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

访问URL地址是（假设没有定义路由的情况下）

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

如果你的控制器是`HelloWorld`，并且定义如下：

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

class HelloWorld
{
```

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

控制器类文件的实际位置是

```
application\index\controller\HelloWorld.php
```

访问URL地址是（假设没有定义路由的情况下）

```
http://localhost/index.php/index/hello_world
```

如果你期望通过

```
http://localhost/index.php/index/HelloWorld
```

可以访问，那么必须关闭URL的自动转换设置

```
        // 是否自动转换URL中的控制器和操作名
        'url_convert'            => false,
```

控制器的命名空间

控制器类的所在命名空间为 `app\module\controller` ，其中根命名空间 `app` 为系统默认，并且只能通过环境变量设置更改，例如我们可以在 `.env` 配置文件中设置：

```
APP_NAMESPACE = application
```

则实际的控制器类应该更改定义如下：

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

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

```
}
```

> 只是命名空间改变了，但实际的文件位置和文件名并没有改变。

单一模块控制器

在应用配置文件`app.php`中设置

```
        // 是否支持多模块
        'app_multi_module'       => false,
```

可以启用单一模块，那么控制器的命名空间中不需要模块名了，类的定义就变成了

```
<?php
namespace app\controller;

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

控制器类文件的实际位置则变成

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

渲染输出

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

下面都是有效的输出方式：

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

class Index
{
    public function hello()
    {
        // 输出hello,world!
        return 'hello,world!';
```

```
    }

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

    public function read()
    {
        // 渲染默认模板输出
        return view();
    }

}
```

> 控制器一般不需要任何输出，直接return即可。

输出转换

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

如果控制器定义为：

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

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

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

}
```

当我们设置输出数据格式为JSON：

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

我们访问

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

输出的结果变成：

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

默认情况下，控制器在ajax请求会对返回类型自动转换，默认为json

如果我们控制器定义

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

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

}
```

我们访问

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

输出的结果变成：

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

多级控制器

支持任意层次级别的控制器，并且支持路由，例如：

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

use think\Controller;
```

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

}
```

该控制器类的文件位置为：

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

访问地址可以使用

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

由于URL访问不能访问默认的多级控制器（可能会把多级控制器名误识别为URL后缀），因此建议所有的多级控制器都通过路由定义后访问，如果要在路由定义中使用多级控制器，可以使用：

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

自动定位控制器

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

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

然后定义控制器如下：

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

use think\Controller;

class Blog extends Controller
{
    public function index()
    {
```

```
        return 'index';
    }

}
```

我们就可以直接访问下面的URL地址了：

```
http://serverName/index.php/index/user/Blog
```

控制器初始化

如果你的控制器类继承了系统控制器基类（`\think\Controller`）的话，可以定义控制器初始化方法`initialize`，该方法会在调用控制器的方法之前首先执行，如非必要，不建议直接修改控制器的架构函数。

例如：

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

use think\Controller;

class Index extends Controller
{

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

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

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

> `initialize`方法不需要任何返回值

如果访问

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

会输出

```
init
hello
```

如果访问

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

会输出

```
init
data
```

前置操作

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

```
['except' => '方法名,方法名']
```

表示这些方法不使用前置方法，

```
['only' => '方法名,方法名']
```

表示只有这些方法使用前置方法。

示例如下:

```
<?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
```

跳转和重定向

页面跳转

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

使用方法很简单，举例如下：

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

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

class Index extends Controller
{
    public function index()
    {
        $User   = new User; //实例化User对象
        $result = $User->save($data);
        if ($result) {
            //设置成功后跳转页面的地址，默认的返回页面是$_SERVER['HTTP_REFERER']
            $this->success('新增成功', 'User/list');
        } else {
            //错误页面的默认跳转页面是返回前一页，通常不需要设置
            $this->error('新增失败');
        }
    }
}
```

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

> 默认的等待时间都是3秒

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

```
'thinkphp/tpl/dispatch_jump.tpl'
```

我们可以改变默认的模板：

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

也可以使用项目内部的模板文件

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

模板文件可以使用模板标签，并且可以使用下面的模板变量：

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

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

重定向

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

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

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

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

或者直接重定向到一个指定的外部URL地址，例如：

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

可以在重定向的时候通过session闪存数据传值，例如

```
$this->redirect('News/category', ['cate_id' => 2], 302, ['data' => 'hello
']);
```

使用redirect助手函数还可以实现更多的功能，例如可以记住当前的URL后跳转

```
redirect('News/category')->remember();
```

需要跳转到上次记住的URL的时候使用：

```
redirect()->restore();
```

> 包括 redirect 、 success 和 error 方法在内的url地址参数不需要使用url方法，系统会自动调用url方法，否则会出现重复的url后缀。

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。

下面的例子用空操作功能实现了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
```

> 空操作方法不需要任何参数，如果要获取当前的操作方法名，直接调用当前请求对象来获取，你也可以使用依赖注入（参考请求章节的依赖注入）

空控制器

空控制器的概念是指当系统找不到指定的控制器名称的时候，系统会尝试定位空控制器(`Error`)，利用这个机制我们可以用来定制错误页面和进行URL的优化。

现在我们把前面的需求进一步，把URL由原来的

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

变成

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

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

我们可以给项目定义一个Error控制器类

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

use think\Request;

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

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

接下来，我们就可以在浏览器里面输入

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

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

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

空控制器和空操作还可以同时使用，用以完成更加复杂的操作。

空控制器Error是可以定义的

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

当找不到控制器的时候，就会定位到MyError控制器类进行操作。

分层控制器

访问控制器

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

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

```
<?php
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

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

例如，我们定义Blog事件控制器如下：

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

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

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

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

定义完成后，就可以用下面的方式实例化并调用方法了：

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

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

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

支持跨模块调用，例如：

```
$event = controller('Admin/Blog', 'event');
```

```
echo $event->update(5); // 输出 update:5
```

表示实例化Admin模块的Blog控制器类，并执行update方法。

除了实例化分层控制器外，还可以直接调用分层控制器类的某个方法，例如：

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

也可以使用助手函数`action`实现相同的功能：

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

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

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

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

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

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

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

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

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

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

资源控制器

资源控制器

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

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

或者使用完整的命名空间生成

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

然后你只需要为资源控制器注册一个资源路由：

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

设置后会自动注册7个路由规则，如下：

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

> 关于资源路由的更多内容请参考路由章节

请求

请求对象

当前的请求对象由`think\Request`类负责，在很多场合下并不需要实例化调用，通常使用依赖注入即可。在其它场合（例如模板输出等）则可以使用`think\facade\Request`静态类操作。

请求对象调用

在控制器中通常情况下有两种方式进行依赖注入。

构造方法注入

```
<?php

namespace app\index\controller;

use think\Request;

class Index
{
    /**
     * @var \think\Request Request实例
     */
    protected $request;

    /**
     * 构造方法
     * @param Request $request Request对象
     * @access public
     */
    public function __construct(Request $request)
    {
                $this->request = $request;
    }

    public function index()
    {
                return $this->request->param('name');
    }
}
```

> 如果你继承了系统的控制器基类 think\Controller 的话，系统已经自动完成了请求对象的构造方法注入了，你可以直接使用 $this->request 属性调用当前的请求对象。

```
<?php

namespace app\index\controller;

use think\Controller;

class Index extends Controller
{

    public function index()
    {
                return $this->request->param('name');
    }
}
```

操作方法注入

另外一种选择是在每个方法中使用依赖注入。

```
<?php

namespace app\index\controller;

use think\Controller;
use think\Request;

class Index extends Controller
{

    public function index(Request $request)
    {
                return $request->param('name');
    }
}
```

无论是否继承系统的控制器基类，都可以使用操作方法注入。

> 更多关于依赖注入的内容，请参考后续的依赖注入章节。

Facade调用

在没有使用依赖注入的场合，可以通过`Facade`机制来静态调用请求对象的方法（注意`use`引入的类库区别）。

```
<?php

namespace app\index\controller;

use think\Controller;
use think\facade\Request;

class Index extends Controller
{

    public function index()
    {
                return Request::param('name');
    }
}
```

该方法也同样适用于依赖注入无法使用的场合。

助手函数

为了简化调用，系统还提供了`request`助手函数，可以在任何需要的时候直接调用当前请求对象。

```
<?php

namespace app\index\controller;

use think\Controller;

class Index extends Controller
{

    public function index()
    {
        return request()->param('name');
    }
}
```

请求信息

请求信息

`Request`对象支持获取当前的请求信息，包括：

方法	含义
`host`	当前访问域名或者IP
`scheme`	当前访问协议
`port`	当前访问的端口
`remotePort`	当前请求的REMOTE_PORT
`protocol`	当前请求的SERVER_PROTOCOL
`contentType`	当前请求的CONTENT_TYPE
`domain`	当前包含协议的域名
`subDomain`	当前访问的子域名
`panDomain`	当前访问的泛域名
`url`	当前完整URL
`baseUrl`	当前URL（不含QUERY_STRING）
`query`	当前请求的QUERY_STRING参数
`baseFile`	当前执行的文件
`root`	URL访问根地址
`rootUrl`	URL访问根目录
`pathinfo`	当前请求URL的pathinfo信息（含URL后缀）
`path`	请求URL的pathinfo信息(不含URL后缀)
`ext`	当前URL的访问后缀
`time`	获取当前请求的时间
`type`	当前请求的资源类型
`method`	当前请求类型

对于上面的这些请求方法，一般调用无需任何参数，但某些方法可以传入`true`参数，表示获取带域名的完整地址，例如：

```
use think\facade\Request;
// 获取完整URL地址 不带域名
Request::url();
// 获取完整URL地址 包含域名
Request::url(true);
```

```
// 获取当前URL（不含QUERY_STRING） 不带域名
Request::baseFile();
// 获取当前URL（不含QUERY_STRING） 包含域名
Request::baseFile(true);
// 获取URL访问根地址 不带域名
Request::root();
// 获取URL访问根地址 包含域名
Request::root(true);
```

> 注意`domain`方法的值本身就包含协议和域名

输入变量

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

为了方便说明，本篇内容的所有示例代码均使用 `Facade` 方式，因此需要首先引入

```
use think\facade\Request;
```

如果你使用的是依赖注入，请自行调整代码为动态调用即可。

主要内容包括：

检测变量是否设置

可以使用 `has` 方法来检测一个变量参数是否设置，如下：

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

变量检测可以支持所有支持的系统变量。

变量获取

变量获取使用 `\think\Request` 类的如下方法及参数：

> 变量类型方法('变量名/变量修饰符','默认值','过滤方法')

变量类型方法包括：

方法	描述
param	获取当前请求的变量
get	获取 $_GET 变量
post	获取 $_POST 变量
put	获取 PUT 变量
delete	获取 DELETE 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
route	获取 路由（包括PATHINFO）变量
file	获取 $_FILES 变量

获取 PARAM 变量

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

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

> param方法会把当前请求类型的参数和路由变量以及GET请求合并，并且路由变量是优先的。

其它的输入变量获取方法用法基本一致，以 post 方法为例主要有如下三种用法。

```
Request::post('name'); // 获取某个post变量
Request::post(); // 获取经过全局过滤的全部post变量
Request::post(false); // 获取全部的post原始（未经全局过滤）变量
```

你无法使用get方法获取路由变量，例如当访问地址是

```
http://localhost/index.php/index/index/hello/name/thinkphp
```

下面的用法是错误的

```
echo Request::get('name'); // 输出为空
```

正确的用法是

```
echo Request::param('name'); // 输出thinkphp
echo Request::route('name'); // 输出thinkphp
```

> 变量名区分大小写，除了 server 和 env 方法的变量名不区分大小写（会自动转为大写后获取）。

默认值

获取输入变量的时候，可以支持默认值，例如当URL中不包含 `$_GET['name']` 的时候，使用下面的方式输出的结果比较。

```
Request::get('name'); // 返回值为null
Request::get('name',''); // 返回值为空字符串
Request::get('name','default'); // 返回值为default
```

前面提到的方法都支持在第二个参数中传入默认值的方式。

变量过滤

> 框架默认没有设置任何全局过滤规则，你可以在应用配置文件中设置全局的过滤规则：
>
> ```
> // 默认全局过滤方法 用逗号分隔多个
> 'default_filter'         => 'htmlspecialchars',
> ```

也支持使用 `Request` 对象进行全局变量的获取过滤，过滤方式包括函数、方法过滤，以及PHP内置的Types of filters，我们可以设置全局变量过滤方法，支持设置多个过滤方法，例如：

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

也可以在获取变量的时候添加过滤方法，例如：

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

可以支持传入多个过滤规则，例如：

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

Request对象还支持PHP内置提供的Filter ID过滤，例如：

```
Request::post('email','',FILTER_VALIDATE_EMAIL);
```

框架对FilterID做了转换支持，因此也可以使用字符串的方式，例如：

```
Request::post('email','','email');
```

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

具体的字符串根据 `filter_list` 函数的返回值来定义。

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

例如，

```
Request::post('email','',FILTER_VALIDATE_EMAIL);
```

就表示，如果不是规范的 `email` 地址的话 返回空字符串。

如果当前不需要进行任何过滤的话，可以使用

```
// 获取get变量 并且不进行任何过滤 即使设置了全局过滤
Request::get('name', '', null);
```

获取部分变量

如果你只需要获取当前请求的部分参数，可以使用：

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

或者使用数组方式

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

> 采用 only 方法能够安全的获取你需要的变量，避免额外变量影响数据处理和写入。

`V5.1.3+` 版本开始，only方法可以支持批量设置默认值，如下：

```
// 设置默认值
Request::only(['id'=>0,'name'=>'']);
```

表示 `id` 的默认值为0， `name` 的默认值为空字符串。

默认获取的是当前请求参数（ `PARAM` 类型变量），如果需要获取其它类型的参数，可以在第二个参数传入，例如：

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

也支持排除某些变量后获取，例如

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

或者使用数组方式

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

同样支持指定变量类型获取：

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

变量修饰符

支持对变量使用修饰符功能，可以一定程度上简单过滤变量，更为严格的过滤请使用前面提过的变量过滤功能。

用法如下：

> Request::变量类型('变量名/修饰符');

默认的变量修饰符是 `/s` ，因此默认的单个变量的取值返回的都是字符串，如果需要传入字符串之外的变量可以使用下面的修饰符，包括：

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

下面是一些例子：

```
Request::get('id/d');
Request::post('name/s');
Request::post('ids/a');
```

> 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。

修改变量

如果需要更改或者额外增加请求变量的值，可以通过下面的方式：

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

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

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

> 更改当前请求类型对应的变量参数后， `param` 方法的参数值会自动更新。

助手函数

为了简化使用，还可以使用系统提供的 `input` 助手函数完成上述大部分功能。

判断变量是否定义

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

获取PARAM参数

```
input('param.name'); // 获取单个参数
input('param.'); // 获取全部参数
// 下面是等效的
input('name');
input('');
```

获取GET参数

```
// 获取单个变量
input('get.id');
// 使用过滤方法获取 默认为空字符串
input('get.name');
// 获取全部变量
input('get.');
```

使用过滤方法

```
input('get.name','','htmlspecialchars'); // 获取get变量 并用htmlspecialchar
s函数过滤
input('username','','strip_tags'); // 获取param变量 并用strip_tags函数过滤
input('post.name','','org\Filter::safeHtml'); // 获取post变量 并用org\Filte
r类的safeHtml方法过滤
```

使用变量修饰符

```
input('get.id/d');
input('post.name/s');
input('post.ids/a');
```

请求类型

获取请求类型

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

请求对象`Request`类提供了下列方法来获取或判断当前请求类型：

用途	方法
获取当前请求类型	method
判断是否GET请求	isGet
判断是否POST请求	isPost
判断是否PUT请求	isPut
判断是否DELETE请求	isDelete
判断是否AJAX请求	isAjax
判断是否PJAX请求	isPjax
判断是否手机访问	isMobile
判断是否HEAD请求	isHead
判断是否PATCH请求	isPatch
判断是否OPTIONS请求	isOptions
判断是否为CLI执行	isCli
判断是否为CGI模式	isCgi

> `method`方法返回的请求类型始终是大写，这些方法都不需要传入任何参数。

> 没有必要在控制器中判断请求类型再来执行不同的逻辑，完全可以在路由中进行设置。

请求类型伪装

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

```
<form method="post" action="">
    <input type="text" name="name" value="Hello">
```

```
    <input type="hidden" name="_method" value="PUT" >
    <input type="submit" value="提交">
</form>
```

提交后的请求类型会被系统识别为 `PUT` 请求。

> 你可以设置为任何合法的请求类型，包括 `GET` 、 `POST` 、 `PUT` 和 `DELETE` 等，但伪装变量 `_method` 只能通过POST请求进行提交。

如果要获取原始的请求类型，可以使用

```
Request::method(true);
```

> 在命令行下面执行的话，请求类型返回的始终是 `GET` 。

如果你需要改变伪装请求的变量名，可以修改应用配置文件：

```
// 表单请求类型伪装变量
'var_method'             => '_m',
```

AJAX/PJAX 伪装

可以对请求进行 `AJAX` 请求伪装，如下：

```
http://localhost/index?_ajax=1
```

或者 `PJAX` 请求伪装

```
http://localhost/index?_pjax=1
```

如果你需要改变伪装请求的变量名，可以修改应用配置文件：

```
// 表单ajax伪装变量
'var_ajax'               => '_a',
// 表单pjax伪装变量
'var_pjax'               => '_p',
```

> `_ajax` 和 `_pjax` 可以通过 `GET/POST/PUT` 等请求变量伪装。

HTTP头信息

可以使用 `Request` 对象的 `header` 方法获取当前请求的 `HTTP` 请求头信息，例如：

```
$info = Request::header();
echo $info['accept'];
echo $info['accept-encoding'];
echo $info['user-agent'];
```

也可以直接获取某个请求头信息，例如：

```
$agent = Request::header('user-agent');
```

`HTTP` 请求头信息的名称不区分大小写，并且 `_` 会自动转换为 `-` ，所以下面的写法都是等效的：

```
$agent = Request::header('user-agent');
$agent = Request::header('User-Agent');
$agent = Request::header('User_Agent');
$agent = Request::header('USER_AGENT');
```

伪静态

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

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

的话，我们可以把下面的URL

```
http://serverName/Home/Blog/read/id/1
```

变成

```
http://serverName/Home/Blog/read/id/1.shtml
```

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

默认情况下，伪静态的设置为`html`，如果我们设置伪静态后缀为空字符串，

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

则支持所有的静态后缀访问，如果要获取当前的伪静态后缀，可以使用`Request`对象的`ext`方法。

例如：

```
http://serverName/index/blog/3.html
http://serverName/index/blog/3.shtml
http://serverName/index/blog/3.xml
http://serverName/index/blog/3.pdf
```

都可以正常访问。

我们可以在控制器的操作方法中获取当前访问的伪静态后缀，例如：

```
$ext = Request::ext();
```

如果希望支持多个伪静态后缀，可以直接设置如下：

```
// 多个伪静态后缀设置 用|分割
'url_html_suffix' => 'html|shtml|xml'
```

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

如果要关闭伪静态访问，可以设置

```
// 关闭伪静态后缀访问
'url_html_suffix' => false,
```

关闭伪静态访问后，不再支持伪静态方式的URL访问，并且伪静态后缀将会被解析为最后一个参数的值，例如：

```
http://serverName/index/blog/read/id/3.html
```

最终的id参数的值将会变成 `3.html`。

参数绑定

参数绑定是把当前请求的变量作为操作方法（也包括架构方法）的参数直接传入，参数绑定并不区分请求类型。

> 参数绑定传入的值会经过全局过滤，如果你有额外的过滤需求可以在操作方法中单独处理。

按名称绑定

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

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

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

    public function archive($year, $month='01')
    {
        return 'year='.$year.'&month='.$month;
    }
}
```

> 注意这里的操作方法并没有具体的业务逻辑，只是简单的示范。

URL的访问地址分别是：

```
http://serverName/index.php/index/blog/read/id/5
http://serverName/index.php/index/blog/archive/year/2016/month/06
```

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

> 变量名绑定不一定由访问URL决定，路由地址也能起到相同的作用

输出的结果依次是：

```
id=5
year=2016&month=06
```

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

```
http://serverName/index.php/index/blog/archive/month/06/year/2016
```

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

如果用户访问的URL地址是（至于为什么会这么访问暂且不提）：

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

那么会抛出下面的异常提示：`参数错误:id`

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

```
    public function read($id=0)
    {
        return 'id='.$id;
    }
```

这样，当我们访问 `http://serverName/index.php/index/blog/read/` 的时候 就会输出

```
id=0
```

> 始终给操作方法的参数定义默认值是一个避免报错的好办法（依赖注入参数除外）

按顺序绑定

还可以支持按照URL的参数顺序进行绑定的方式，合理规划URL参数的顺序绑定对简化URL地址可以起到一定的帮助。

还是上面的例子，控制器不变，还是使用：

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

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

    public function archive($year='2016',$month='01')
    {
        return 'year='.$year.'&month='.$month;
    }
}
```

我们在配置文件中添加配置参数如下：

```
// URL参数方式改成顺序解析
'url_param_type'         => 1,
```

接下来，访问下面的URL地址：

```
http://serverName/index.php/index/blog/read/5
http://serverName/index.php/index/blog/archive/2016/06
```

输出的结果依次是：

```
id=5
year=2016&month=06
```

按参数顺序绑定的话，参数的顺序不能随意调整，如果访问：

```
http://serverName/index.php/index/blog/archive/06/2016
```

最后的输出结果则变成：

```
id=5
year=06&month=2016
```

> 按顺序绑定参数的话，操作方法的参数只能使用路由变量或者PATHINFO变量，而不能使用get或者post变量。

请求缓存

请求缓存

支持请求缓存功能，支持对GET请求设置缓存访问，并设置有效期。

> 请求缓存仅对GET请求有效

有两种方式可以设置请求缓存：

路由设置

可以在路由规则里面调用`cache`方法设置当前路由规则的请求缓存，例如：

```
// 定义GET请求路由规则 并设置3600秒的缓存
Route::get('new/:id','News/read')->cache(3600);
```

第二次访问相同的路由地址的时候，会自动获取请求缓存的数据响应输出，并发送`304`状态码。

默认请求缓存的标识为当前访问的`pathinfo`地址，可以定义请求缓存的标识，如下：

```
// 定义GET请求路由规则 并设置3600秒的缓存
Route::get('new/:id','News/read')->cache(
        [
                'new/:id/:page', 3600
        ]
);
```

`:id`、`:page`表示使用当前请求的`param`参数进行动态标识替换，也就是根据`id`和`page`变量进行`3600`秒的请求缓存。

> 如果 cache 参数传入 false ，则表示关闭当前路由的请求缓存（即使开启全局请求缓存）。

支持给一组路由设置缓存标签

```
// 定义GET请求路由规则 并设置3600秒的缓存
Route::get('new/:id','News/read')->cache(
```

```
        [
                'new/:id/:page', 3600, 'page'
        ]
);
```

这样可以在需要的时候统一清理缓存标签为 `page` 的请求缓存。

动态设置

可以在公共文件或者行为中动态设置请求缓存，例如：

```
Request::cache('blog/:id',3600);
```

表示对 `blog/:id` 定义的动态访问地址进行 `3600` 秒的请求缓存。

> 变量支持当前的请求变量（也就是 `param` 方法的所有变量）。

可以使用当前的URL地址作为缓存标识，如下：

```
Request::cache('__URL__',600);
```

支持对某个URL后缀的请求进行缓存，例如：

```
Request::cache('[html]',600);
```

表示对所有的 `html` 后缀访问（GET）请求进行10分钟的缓存。

也支持设置请求缓存标签，例如：

```
Request::cache('blog/:id/:page',3600, 'page');
```

自动缓存

可以通过开启自动缓存和全局缓存有效期设置请求缓存，支持在模块配置中单独设置开启请求缓存。

只需要在配置文件中开启：

```
'request_cache' =>      true,
'request_cache_expire'  =>      3600,
```

就会自动根据当前请求URL地址（只针对GET请求类型）进行请求缓存，全局缓存有效期为3600秒。

如果需要对全局缓存设置缓存规则，可以直接设置`request_cache`参数为字符串，例如下面的方式：

```
'request_cache' =>      '__URL__',
'request_cache_expire'  =>      3600,
```

缓存标识支持下面的特殊定义

标识	含义
`__MODULE__`	当前模块名
`__CONTROLLER__`	当前控制器名
`__ACTION__`	当前操作名
`__URL__`	当前完整URL地址（包含域名）

全局请求缓存支持设置排除规则，使用方法如下：

```
    'request_cache'         => true,
    'request_cache_expire' => 3600,
    'request_cache_except' => [
        '/blog/index',
        '/user/member',
    ],
```

排除规则为不使用请求缓存的地址（不支持变量）开头部分（不区分大小写）。

> 路由中设置的请求缓存依然有效并且优先，如果需要设置特殊的请求缓存有效期就可以直接在路由中设置。

响应

响应

> 响应（ Response ）对象用于动态响应客户端请求，控制发送给用户的信息。通常用于输出数据给客户端或者浏览器。

`ThinkPHP5.1`的`Response`响应对象由`think\Response`类或者子类完成，ThinkPHP的响应输出是自动的（命令行模式除外），最终会调用`Response`对象的`send`方法完成输出。

响应输出

响应输出

大多数情况，我们不需要关注`Response`对象本身，只需要在控制器的操作方法中返回数据即可，系统会根据`default_return_type`和`default_ajax_return`配置自动决定响应输出的类型。

默认的自动响应输出会自动判断是否`AJAX`请求，如果是的话会自动输出`default_ajax_return`配置的输出类型。

最简单的响应输出是直接在路由闭包或者控制器操作方法中返回一个字符串，例如：

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

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

class Index
{
    public function hello($name='thinkphp')
    {
        return 'Hello,' . $name . '!';
    }
}
```

由于默认是输出`Html`输出，所以直接以html页面方式输出响应内容。

如果修改配置文件，设置：

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

则访问的输出结果就变成了`JSON`字符串（同样，还可以修改输出类型为`xml`）。

为了规范和清晰起见，最佳的方式是在控制器最后明确输出类型（毕竟一个确定的请求是有明确的响应输出类型），默认支持的输出类型包括：

输出类型	快捷方法	对应Response类

HTML输出	response	\think\Response
渲染模板输出	view	\think\response\View
JSON输出	json	\think\response\Json
JSONP输出	jsonp	\think\response\Jsonp
XML输出	xml	\think\response\Xml
页面重定向	redirect	\think\response\Redirect

每一种输出类型其实对应了一个不同的`Response`子类（`response()`函数对应的是`Response`基类），也可以在应用中自定义`Response`子类满足特殊需求的输出。

例如我们需要输出一个JSON数据给客户端（或者AJAX请求），可以使用：

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

class Index
{
    public function hello()
    {
        $data = ['name' => 'thinkphp', 'status' => '1'];
        return json($data);
    }
}
```

> 这些助手函数的返回值都是`Response`类或者子类的对象实例，所以后续可以调用`Response`基类或者当前子类的相关方法，后面我们会讲解相关方法。

如果你只需要输出一个html格式的内容，可以直接使用

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

class Index
{
    public function hello()
    {
        $data = 'Hello,ThinkPHP!';
        return response($data);
    }
}
```

或者使用`return`直接返回输出的字符串（前提是你的`default_return_type`设置是

`html`)。

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

class Index
{
    public function hello()
    {
        return 'Hello,ThinkPHP!';
    }
}
```

响应参数

`Response`对象提供了一系列方法用于设置响应参数，包括设置输出内容、状态码及header信息等，并且支持链式调用以及多次调用。

设置数据

`Response`基类提供了`data`方法用于设置响应数据。

```
response()->data($data);
json()->data($data);
```

不过需要注意的是`data`方法设置的只是原始数据，并不一定是最终的输出数据，最终的响应输出数据是会根据当前的`Response`响应类型做自动转换的，例如：

```
json()->data($data);
```

最终的输出数据就是`json_encode($data)`转换后的数据。

如果要获取当前响应对象实例的实际输出数据可以使用`getContent`方法。

设置状态码

`Response`基类提供了`code`方法用于设置响应数据，但大部分情况一般我们是直接在调用助手函数的时候直接传入状态码，例如：

```
json($data,201);
view($data,401);
```

或者在后面链式调用`code`方法是等效的：

```
json($data)->code(201);
```

> 除了 redirect 函数的默认返回状态码是 302 之外，其它方法没有指定状态码都是返回 200 状态码。

如果要获取当前响应对象实例的状态码的值，可以使用`getCode`方法。

设置头信息

可以使用 `Response` 类的 `header` 设置响应的头信息

```
json($data)->code(201)->header(['Cache-control' => 'no-cache,must-revalid
ate']);
```

`header` 方法支持两种方式设置，如果传入数组，则表示批量设置，如果传入两个参数，第一个参数表示头信息名，第二个参数表示头信息的值，例如：

```
// 单个设置
header('Cache-control', 'no-cache,must-revalidate');
// 批量设置
header([
    'Cache-control' => 'no-cache,must-revalidate',
    'Last-Modified' => gmdate('D, d M Y H:i:s') . ' GMT',
]);
```

除了 `header` 方法之外， `Response` 基类还提供了常用头信息的快捷设置方法：

方法名	作用
`lastModified`	设置 `Last-Modified` 头信息
`expires`	设置 `Expires` 头信息
`eTag`	设置 `ETag` 头信息
`cacheControl`	设置 `Cache-control` 头信息
`contentType`	设置 `Content-Type` 头信息

除非你要清楚自己在做什么，否则不要随便更改这些头信息，每个 `Response` 子类都有默认的 `contentType` 信息，一般无需设置。

你可以使用 `getHeader` 方法获取当前响应对象实例的头信息。

设置额外参数

有些时候，响应输出需要设置一些额外的参数，例如：
在进行 `json` 输出的时候需要设置 `json_encode` 方法的额外参数， `jsonp` 输出的时候需要设置 `jsonp_handler` 等参数，这些都可以使用 `options` 方法来进行处理，例如：

```
json($data)
    ->options('json_encode_param',JSON_PRETTY_PRINT);
```

也可以支持传入数组作为参数：

```
jsonp($data)
    ->options([
        'var_jsonp_handler'     => 'callback',
        'default_jsonp_handler' => 'jsonpReturn',
        'json_encode_param'     => JSON_PRETTY_PRINT,
    ]);
```

关闭当前的请求缓存（ V5.1.5+ ）

V5.1.5+ 版本开始，支持使用 allowCache 方法动态控制是否需要使用请求缓存。

```
// 关闭当前页面的请求缓存
json($data)->code(201)->allowCache(false);
```

自定义响应

如果需要特别的自定义响应输出，可以自定义一个 Response 子类，并且在控制器的操作方法中直接返回。又或者通过设置响应参数的方式进行响应设置输出。

重定向

重定向

可以使用`redirect`助手函数进行重定向

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

class Index
{
    public function hello()
    {
        return redirect('http://www.thinkphp.cn');
    }
}
```

> `redirect`函数和控制器的`redirect`方法的参数顺序有所区别

重定向传参

如果是站内重定向的话，可以支持URL组装，有两种方式组装URL，第一种是直接使用完整地址（`/`打头）

```
redirect('/index/index/hello/name/thinkphp');
```

这种方式会保持原来地址不做任何转换，第二种方式是使用`params`方法配合，例如：

```
redirect('hello')->params(['name'=>'thinkphp']);
```

最终重定向的URL地址和前面的一样的，系统内部会自动判断并调用`url`（用于快速生成URL地址的助手函数）方法进行地址生成，或者使用下面的方法

```
redirect('hello',['name'=>'thinkphp']);
```

还可以支持使用`with`方法附加`Session`闪存数据重定向。

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

```
class Index
{
    public function index()
    {
        return redirect('hello')->with('name','thinkphp');
    }

    public function hello()
    {
        $name = session('name');
        return 'hello,'.$name.'!';
    }
}
```

从示例可以看到重定向隐式传值使用的是 `Session` 闪存数据隐式传值，并且仅在下一次请求有效，再次访问重定向地址的时候无效。

记住请求地址

在很多时候，我们重定向的时候需要记住当前请求地址（为了便于跳转回来），我们可以使用 `remember` 方法记住重定向之前的请求地址。

下面是一个示例，我们第一次访问 `index` 操作的时候会重定向到 `hello` 操作并记住当前请求地址，然后操作完成后到 `restore` 方法， `restore` 方法则会自动重定向到之前记住的请求地址，完成一次重定向的回归，回到原点！（再次刷新页面又可以继续执行）

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

class Index
{
    public function index()
    {
        // 判断session完成标记是否存在
        if (session('?complete')) {
            // 删除session
            session('complete', null);
            return '重定向完成，回到原点!';
        } else {
            // 记住当前地址并重定向
            return redirect('hello')
                ->with('name', 'thinkphp')
                ->remember();
        }
    }

    public function hello()
    {
        $name = session('name');
```

```
        return 'hello,' . $name . '! <br/><a href="/index/index/restore">
点击回到来源地址</a>';
    }

    public function restore()
    {
        // 设置session标记完成
        session('complete', true);
        // 跳回之前的来源地址
        return redirect()->restore();
    }
}
```

数据库

连接数据库

> ThinkPHP内置了抽象数据库访问层，把不同的数据库操作封装起来，我们只需要使用公共的Db类进行操作，而无需针对不同的数据库写不同的代码和底层实现，Db类会自动调用相应的数据库驱动来处理。数据库抽象访问层基于PDO方式，目前内置包含了Mysql、SqlServer、PgSQL、Sqlite等数据库的支持。

如果应用需要使用数据库，必须配置数据库连接信息，数据库的配置文件有多种定义方式。

配置文件

在应用配置目录或者模块配置目录（不清楚配置目录位置的话参考配置章节）下面的`database.php`中（后面统称为数据库配置文件）配置下面的数据库参数：

```
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'      => 'think_',
];
```

系统默认支持的数据库`type`包括：

type	数据库
mysql	MySQL
sqlite	SqLite
pgsql	PgSQL
sqlsrv	SqlServer

`type`参数支持命名空间完整定义，不带命名空间定义的话，默认采用`\think\db\connector`作为命名空间，如果使用应用自己扩展的数据库驱动，可以配置为：

```
// 数据库类型
'type'        => '\org\db\Mysql',
```

表示数据库的连接器采用`\org\db\Mysql`类作为数据库连接驱动，而不是默认的`\think\db\connector\Mysql`。

每个模块可以设置独立的数据库连接参数，并且相同的配置参数可以无需重复设置，例如我们可以在`admin`模块的数据库配置文件中定义：

```
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。

ThinkPHP`5.1`支持数据库的断线重连机制（默认关闭），需要的话，在数据库配置文件中设置：

```
// 开启断线重连
'break_reconnect' => true,
```

连接参数

可以针对不同的连接需要添加数据库的连接参数（具体的连接参数可以参考PHP手册），内置采用的参数包括如下：

```
PDO::ATTR_CASE              => PDO::CASE_NATURAL,
PDO::ATTR_ERRMODE           => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_ORACLE_NULLS      => PDO::NULL_NATURAL,
PDO::ATTR_STRINGIFY_FETCHES => false,
PDO::ATTR_EMULATE_PREPARES  => false,
```

在数据库配置文件中设置的`params`参数中的连接配置将会和内置的设置参数合并，如果需

要使用长连接，并且返回数据库的小写列名，可以在数据库配置文件中增加下面的定义：

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

> 你可以在`params`参数里面配置任何PDO支持的连接参数。

方法配置

我们可以调用`Db::connect`方法动态配置数据库连接信息，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');
```

> `connect`方法必须在查询的最开始调用，否则可能会导致部分查询失效。

字符串DSN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

> 字符串方式可能无法定义某些参数，例如前缀和连接参数。

实际应用中，我们应当尽量避免把数据库配置信息写在代码中，而应该统一定义在配置文件，我们可以在数据库配置文件中增加额外的配置参数，例如：

```
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'      => 'think_',
    //数据库配置1
    'db_config1' => [
        // 数据库类型
        'type'        => 'mysql',
        // 服务器地址
        'hostname'    => '192.168.1.8',
        // 数据库名
        'database'    => 'thinkphp',
        // 数据库用户名
        'username'    => 'root',
        // 数据库密码
        'password'    => '1234',
        // 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@192.168.1.10:3306/thinkphp#utf8',
];
```

然后需要动态链接的时候使用下面的方式

```
Db::connect('db_config1');
Db::connect('db_config2');
```

动态连接数据库的`connect`方法仅对当次查询有效。

> 这种方式的动态连接和切换数据库比较方便，经常用于多数据库连接的应用需求。

模型类定义

如果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按照给定的数据库配置进行连接，而不是配置文件中设置的默认连接信息，例如：

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

use think\Model;

class User extends Model
{
    // 直接使用配置参数名
    protected $connection = 'db_config1';
}
```

和前面一种方法配置一样，`connection`属性可以支持数组和字符串方式。

> 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。

数据库调试模式

无论应用是否部署模式，数据库有自己独立的调试模式开关，数据库配置参数中的`debug`参数就是数据库调试模式的开关（默认关闭）。

```
// 数据库调试模式
'debug'           => true,
```

数据库调试模式开启后，可以支持下列行为：

- 记录SQL日志；
- 分析SQL性能；
- 支持SQL监听；

由于上述行为不可避免会产生额外的开销，因此对性能存在一定的影响，但并不大，因为所有的日志是最终统一一次性写入，而且可以设置为某个用户才写入日志。

> 在生产模式下面，必须关闭应用调试模式（ `app_debug` ），否则会暴露你的服务器敏感信息。和应用调试模式不同，开启数据库调试模式并不会对外暴露任何安全信息，因此是否开启数据库调试模式，看自己的需求。

配置参数参考

下面是默认支持的数据库连接信息：

参数名	描述	默认值
type	数据库类型	无
hostname	数据库地址	127.0.0.1
database	数据库名称	无
username	数据库用户名	无
password	数据库密码	无
hostport	数据库端口号	无
dsn	数据库连接dsn信息	无
params	数据库连接参数	空
charset	数据库编码	utf8
prefix	数据库的表前缀	无
debug	是否调试模式	false
deploy	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)	0
rw_separate	数据库读写是否分离 主从式有效	false
master_num	读写分离后 主服务器数量	1
slave_no	指定从服务器序号	无
fields_strict	是否严格检查字段是否存在	true
resultset_type	数据集返回类型	array
auto_timestamp	自动写入时间戳字段	false
sql_explain	是否需要进行SQL性能分析 开启调试有效	false
query	指定查询对象	think\db\Query

常用数据库连接参数（ `params` ）可以参考PHP在线手册中的以 `PDO::ATTR_` 开头的常量。

> 注意：
>
> >***
> 如果是使用 pgsql 数据库驱动的话，请先导入 thinkphp/library/think/db/connector/pgsql.sql 文件到数据库执行。

查询构造器

查询数据

基本查询

查询单个数据使用`find`方法：

```
// table方法必须指定完整的数据表名
Db::table('think_user')->where('id',1)->find();
```

最终生成的SQL语句可能是：

```
SELECT * FROM `think_user` WHERE  `id` = 1 LIMIT 1
```

> find 方法查询结果不存在，返回 null，否则返回结果数组

如果希望在没有找到数据后抛出异常可以使用

```
// table方法必须指定完整的数据表名
Db::table('think_user')->where('id',1)->findOrFail();
```

如果没有查找到数据，则会抛出一个`think\db\exception\DataNotFoundException`异常。

查询多个数据（数据集）使用`select`方法：

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

最终生成的SQL语句可能是：

```
SELECT * FROM `think_user` WHERE `status` = 1
```

> select 方法查询结果是一个二维数组，如果结果不存在，返回空数组

如果希望在没有查找到数据后抛出异常可以使用

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

如果没有查找到数据，同样也会抛出一个`think\db\exception\DataNotFoundException`异常。

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();
```

> 如果你的数据表没有设置表前缀的话，那么 `name` 和 `table` 方法效果一致。

在`find`和`select`方法之前可以使用所有的链式操作（参考链式操作章节）方法。

> 默认情况下，`find`和`select`方法返回的都是数组，区别在于后者是二维数组。

助手函数

系统提供了一个`db`助手函数，可以更方便的查询：

```
db('user')->where('id',1)->find();
db('user')->where('status',1)->select();
```

`db`方法的第一个参数的作用和`name`方法一样，如果需要使用不同的数据库连接，可以使用：

```
db('user','db_config1')->where('id', 1)->find();
```

值和列查询

查询某个字段的值可以用

```
// 返回某个字段的值
Db::table('think_user')->where('id',1)->value('name');
```

> value 方法查询结果不存在，返回 null

查询某一列的值可以用

```
// 返回数组
Db::table('think_user')->where('status',1)->column('name');
// 指定id字段的值作为索引
Db::table('think_user')->where('status',1)->column('name','id');
```

如果要返回完整数据，并且添加一个索引值的话，可以使用

```
// 指定id字段的值作为索引 返回所有数据
Db::table('think_user')->where('status',1)->column('*','id');
```

> column 方法查询结果不存在，返回空数组

数据分批处理

如果你需要处理成千上百条数据库记录，可以考虑使用 `chunk` 方法，该方法一次获取结果集的一小块，然后填充每一小块数据到要处理的闭包，该方法在编写处理大量数据库记录的时候非常有用。

比如，我们可以全部用户表数据进行分批处理，每次处理 100 个用户记录：

```
Db::table('think_user')->chunk(100, function($users) {
    foreach ($users as $user) {
        //
    }
});
// 或者交给回调方法myUserIterator处理
Db::table('think_user')->chunk(100, 'myUserIterator');
```

你可以通过从闭包函数中返回 `false` 来中止对后续数据集的处理：

```
Db::table('think_user')->chunk(100, function($users) {
    foreach ($users as $user) {
        // 处理结果集...
                if($user->status==0){
            return false;
        }
    }
});
```

也支持在 `chunk` 方法之前调用其它的查询方法，例如：

```
Db::table('think_user')
->where('score','>',80)
->chunk(100, function($users) {
    foreach ($users as $user) {
        //
    }
});
```

`chunk` 方法的处理默认是根据主键查询，支持指定字段，例如：

```
Db::table('think_user')->chunk(100, function($users) {
    // 处理结果集...
    return false;
},'create_time');
```

并且支持指定处理数据的顺序。

```
Db::table('think_user')->chunk(100, function($users) {
    // 处理结果集...
    return false;
},'create_time', 'desc');
```

> chunk 方法一般用于命令行操作批处理数据库的数据，不适合WEB访问处理大量数据，很容易导致超时。

大批量数据处理

如果你需要处理大量的数据，可以使用新版提供的游标查询功能，该查询方式利用了PHP的生成器特性，可以大幅减少大量数据查询的内存占用问题。

```
$cursor = Db::table('user')->where('status', 1)->cursor();
foreach($cursor as $user){
        echo $user['name'];
}
```

`cursor` 方法返回的是一个生成器对象，`user` 变量是数据表的一条数据（数组）。

JSON类型数据查询（ `mysql` ）

```
// 查询JSON类型字段 （info字段为json类型）
Db::table('think_user')
        ->where('info->email','thinkphp@qq.com')
```

```
            ->find();
```

添加数据

添加一条数据

使用 `Db` 类的 `insert` 方法向数据库提交数据

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->insert($data);
```

> `insert` 方法添加数据成功返回添加成功的条数，通常情况返回 1

或者使用 `data` 方法配合 `insert` 使用。

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')
    ->data($data)
    ->insert();
```

> 如果你的数据表里面没有 `foo` 或者 `bar` 字段，那么就会抛出异常。

如果不希望抛出异常，可以使用下面的方法：

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->strict(false)->insert($data);
```

不存在的字段的值将会直接抛弃。

如果是mysql数据库，支持 `replace` 写入，例如：

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->insert($data, true);
```

添加数据后如果需要返回新增数据的自增主键，可以使用 `insertGetId` 方法新增数据并返回主键值：

```
$userId = Db::name('user')->insertGetId($data);
```

> `insertGetId` 方法添加数据成功返回添加数据的自增主键

添加多条数据

添加多条数据直接向 Db 类的 `insertAll` 方法传入需要添加的数据即可

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
];
Db::name('user')->insertAll($data);
```

> insertAll 方法添加数据成功返回添加成功的条数
>
> 如果是mysql数据库，支持 `replace` 写入，例如：

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
];
Db::name('user')->insertAll($data, true);
```

也可以使用 `data` 方法

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
];
Db::name('user')->data($data)->insertAll();
```

> 确保要批量添加的数据字段是一致的

如果批量插入的数据比较多，可以指定分批插入，使用 `limit` 方法指定每次插入的数量限制。

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
```

```
    ...
];
// 分批写入 每次最多100条数据
Db::name('user')->data($data)->limit(100)->insertAll();
```

更新数据

更新数据

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

实际生成的SQL语句可能是：

```
UPDATE `think_user`  SET `name`='thinkphp'  WHERE  `id` = 1
```

> update 方法返回影响数据的条数，没修改任何数据返回 0

支持使用 `data` 方法传入要更新的数据

```
Db::name('user')
    ->where('id', 1)
    ->data(['name' => 'thinkphp'])
    ->update();
```

> 如果 `update` 方法和 `data` 方法同时传入更新数据，则会进行合并。

如果数据中包含主键，可以直接使用：

```
Db::name('user')
    ->update(['name' => 'thinkphp','id'=>1]);
```

实际生成的SQL语句和前面用法是一样的：

```
UPDATE `think_user`  SET `name`='thinkphp'  WHERE  `id` = 1
```

如果要更新的数据需要使用 `SQL` 函数或者其它字段，可以使用下面的方式：

```
Db::name('user')
    ->where('id', 1)
```

```
    ->update([
        'read_time'           => ['exp','read_time+1'],
        'score'       => ['exp','score-3'],
        'name'        => ['exp','UPPER(name)'],
    ]);
```

实际生成的SQL语句可能是：

```
UPDATE `think_user`  SET `read_time`=read_time+1,`score`=score-3,`name`=U
PPER(name)  WHERE  `id` = 1
```

或者调用`data`、`inc`、`dec`和`exp`方法简化写法，上面的例子可以改为：

```
Db::name('user')
    ->where('id',1)
    ->inc('read_time')
    ->dec('score',3)
    ->exp('name','UPPER(name)')
    ->update();
```

实际生成的SQL语句有所变化：

```
UPDATE `think_user`  SET `read_time` = `read_time` + 1 , `score` = `score
` - 3 , `name` = UPPER(name)  WHERE  `id` = 1
```

一个细小的区别就是后面生成的SQL自动加上了关键词处理，所以建议使用后者进行数据更新，更加安全。

更新字段值

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

最终生成的SQL语句可能如下：

```
UPDATE `think_user`  SET `name` = 'thinkphp'  WHERE  `id` = 1
```

> setField 方法返回影响数据的条数，没修改任何数据字段返回 0

可以使用`setInc/setDec`方法自增或自减一个字段的值（ 如不加第二个参数，默认步长为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);
```

最终生成的SQL语句可能是：

```
UPDATE `think_user`  SET `score` = `score` + 1  WHERE  `id` = 1
UPDATE `think_user`  SET `score` = `score` + 5  WHERE  `id` = 1
UPDATE `think_user`  SET `score` = `score` - 1  WHERE  `id` = 1
UPDATE `think_user`  SET `score` = `score` - 5  WHERE  `id` = 1
```

`setInc/setDec`支持延时更新，如果需要延时更新则传入第三个参数，下例中延时10秒更新。

```
Db::name('user')->where('id', 1)->setInc('score', 1, 10);
```

> setInc/setDec 方法返回影响数据的条数，如果使用了延迟更新的话，可能会返回true

删除数据

删除数据

```
// 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();
```

最终生成的SQL语句可能是：

```
DELETE FROM `think_user` WHERE  `id` = 1
DELETE FROM `think_user` WHERE  `id` IN (1,2,3)
DELETE FROM `think_user` WHERE  `id` = 1
DELETE FROM `think_user` WHERE  `id` < 10
```

> delete 方法返回影响数据的条数，没有删除返回 0

如果不带任何条件调用`delete`方法会提示错误，如果你确实需要删除所有数据，可以使用

```
// 无条件删除所有数据
Db::name('user')->delete(true);
```

最终生成的SQL语句是（删除了表的所有数据）：

```
DELETE FROM `think_user`
```

> 一般情况下，业务数据不建议真实删除数据，系统提供了软删除机制（模型中使用软删除更为方便）。

```
// 软删除数据 使用delete_time字段标记删除
Db::name('user')
        ->where('id', 1)
        ->useSoftDelete('delete_time',time())
    ->delete();
```

实际生成的SQL语句可能如下（执行的是`UPDATE`操作）：

```
UPDATE `think_user`  SET `delete_time` = '1515745214'  WHERE  `id` = 1
```

`useSoftDelete`方法表示使用软删除，并且指定软删除字段为`delete_time`，写入数据为当前的时间戳。

查询表达式

查询表达式

查询表达式支持大部分的SQL查询语法，也是 `ThinkPHP` 查询语言的精髓，查询表达式的使用格式：

```
where('字段名','表达式','查询条件');
whereOr('字段名','表达式','查询条件');
```

`5.1` 还支持新的查询方法

```
whereField('表达式','查询条件');
whereOrField('表达式','查询条件');
```

> `Field` 使用字段的驼峰命名方式。

表达式不分大小写，支持的查询表达式有下面几种：

表达式	含义	快捷查询方法
=	等于	
<>	不等于	
>	大于	
>=	大于等于	
<	小于	
<=	小于等于	
LIKE	模糊查询	`whereLike/whereNotLike`
[NOT] BETWEEN	（不在）区间查询	`whereBetween/whereNotBetween`
[NOT] IN	（不在）IN 查询	`whereIn/whereNotIn`
[NOT] NULL	查询字段是否（不）是NULL	`whereNull/whereNotNull`
[NOT] EXISTS	EXISTS查询	`whereExists/whereNotExists`
[NOT] REGEXP	正则（不）匹配查询（仅支持Mysql）	
[NOT] BETWEEM TIME	时间区间比较	whereBetweenTime

> TIME	大于某个时间	
< TIME	小于某个时间	
>= TIME	大于等于某个时间	
<= TIME	小于等于某个时间	
EXP	表达式查询，支持SQL语法	`whereExp`

表达式查询的用法示例如下：

等于（=）

例如：

```
Db::name('user')->where('id','=',100)->select();
```

和下面的查询等效

```
Db::name('user')->where('id',100)->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` = 100
```

不等于（<>）

例如：

```
Db::name('user')->where('id','<>',100)->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` <> 100
```

大于（>）

例如：

```
Db::name('user')->where('id','>',100)->select();
```

最终生成的SQL语句是：

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` > 100
```

大于等于（>=）

例如：

```
Db::name('user')->where('id','>=',100)->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` >= 100
```

小于（<）

例如：

```
Db::name('user')->where('id','<',100)->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` < 100
```

小于等于（<=）

例如：

```
Db::name('user')->where('id','<=',100)->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `id` <= 100
```

[NOT] LIKE： 同sql的LIKE

例如：

```
Db::name('user')->where('name','like','thinkphp%')->select();
```

最终生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `name` LIKE 'thinkphp%'
```

`like`查询支持使用数组

```
Db::name('user')->where('name','like',['%think','php%'],'OR')->select();
```

实际生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE  (`name` LIKE '%think' OR `name` LIKE 'p
hp%')
```

为了更加方便，应该直接使用`whereLike`方法

```
Db::name('user')->whereLike('name','thinkphp%')->select();
Db::name('user')->whereNotLike('name','thinkphp%')->select();
```

[NOT] BETWEEN ：同sql的[not] between

查询条件支持字符串或者数组，例如：

```
Db::name('user')->where('id','between','1,8')->select();
```

和下面的等效：

```
Db::name('user')->where('id','between',[1,8])->select();
```

最终生成的SQL语句都是：

```
SELECT * FROM `think_user` WHERE  `id` BETWEEN 1 AND 8
```

最快捷的查询方法是：

```
Db::name('user')->whereBetween('id','1,8')->select();
Db::name('user')->whereNotBetween('id','1,8')->select();
```

[NOT] IN： 同sql的[not] in

查询条件支持字符串或者数组，例如：

```
Db::name('user')->where('id','in','1,5,8')->select();
```

和下面的等效：

```
Db::name('user')->where('id','in',[1,5,8])->select();
```

最终的SQL语句为：

```
SELECT * FROM `think_user` WHERE  `id` IN (1,5,8)
```

最快捷的查询方法是：

```
Db::name('user')->whereIn('id','1,5,8')->select();
Db::name('user')->whereNotIn('id','1,5,8')->select();
```

> [NOT] IN 查询支持使用闭包方式

[NOT] NULL ：

查询字段是否（不）是`Null`，例如：

```
Db::name('user')->where('name', null)
->where('email','null')
->where('name','not null')
->select();
```

实际生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE  `name` IS NULL  AND `email` IS NULL  AN
D `name` IS NOT NULL
```

如果你需要查询一个字段的值为字符串`null`或者`not null`，应该使用：

```
Db::name('user')->where('title','=', 'null')
```

```
->where('name','=', 'not null')
->select();
```

推荐的方式是使用`whereNull`和`whereNotNull`方法查询。

```
Db::name('user')->whereNull('name')
->whereNull('email')
->whereNotNull('name')
->select();
```

EXP：表达式

支持更复杂的查询情况 例如：

```
Db::name('user')->where('id','in','1,3,8')->select();
```

可以改成：

```
Db::name('user')->where('id','exp',' IN (1,3,8) ')->select();
```

`exp`查询的条件不会被当成字符串，所以后面的查询条件可以使用任何SQL支持的语法，包括使用函数和字段名称。

动态查询

对于上面的查询表达式，可以使用动态查询方法进行简化，例如：

```
Db::name('user')->where('id','>=',100)->select();
```

可以简化为：

```
Db::name('user')->whereId('>=',100)->select();
```

链式操作

数据库提供的链式操作方法，可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的CURD操作（原生查询不支持链式操作）。

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();
```

每次 `Db` 类的静态方法调用是创建一个新的查询对象实例，如果你需要多次复用使用链式操作值，可以使用下面的方法。

```
$user = Db::table('think_user');
```

```
$user->order('create_time')
    ->where('status',1)
    ->select();

// 会自动带上前面的where条件和order排序的值
$user->where('id', '>', 0)->select();
```

当前查询对象在查询之后仍然会保留链式操作的值，除非你调用`removeOption`方法清空链式操作的值。

```
$user = Db::table('think_user');
$user->order('create_time')
    ->where('status',1)
    ->select();

// 清空where查询条件值 保留其它链式操作
$user->removeOption('where')
        ->where('id', '>', 0)
    ->select();
// 清空所有链式操作的值 重新查询
$user->removeOption()
        ->where('id', '>', 0)
    ->order('status')
    ->select();
```

系统支持的链式操作方法包含：

连贯操作	作用	支持的参数类型
where*	用于AND查询	字符串、数组和对象
whereOr*	用于OR查询	字符串、数组和对象
wheretime*	用于时间日期的快捷查询	字符串
table	用于定义要操作的数据表名称	字符串和数组
alias	用于给当前数据表定义别名	字符串
field*	用于定义要查询的字段（支持字段排除）	字符串和数组
order*	用于对结果排序	字符串和数组
limit	用于限制查询结果数量	字符串和数字
page	用于查询分页（内部会转换成limit）	字符串和数字
group	用于对查询的group支持	字符串
having	用于对查询的having支持	字符串
join*	用于对查询的join支持	字符串和数组
union*	用于对查询的union支持	字符串、数组和对象
view*	用于视图查询	字符串、数组

distinct	用于查询的distinct支持	布尔值
lock	用于数据库的锁机制	布尔值
cache	用于查询缓存	支持多个参数
relation*	用于关联查询	字符串
with*	用于关联预载入	字符串、数组
bind*	用于数据绑定操作	数组或多个参数
comment	用于SQL注释	字符串
force	用于数据集的强制索引	字符串
master	用于设置主服务器读取数据	布尔值
strict	用于设置是否严格检测字段名是否存在	布尔值
sequence	用于设置Pgsql的自增序列名	字符串
failException	用于设置没有查询到数据是否抛出异常	布尔值
partition	用于设置分表信息	数组 字符串

> 所有的连贯操作都返回当前的模型实例对象（this），其中带*标识的表示支持多次调用。

where

`where`方法的用法是ThinkPHP查询语言的精髓，也是ThinkPHP `ORM`的重要组成部分和亮点所在，可以完成包括普通查询、表达式查询、快捷查询、区间查询、组合查询在内的查询操作。`where`方法的参数支持的变量类型包括字符串、数组和闭包。

> 和`where`方法相同用法的方法还包括`whereOr`、`whereIn`等一系列快捷查询方法，下面仅以`where`为例说明用法。

表达式查询

> 表达式查询是官方推荐使用的查询方式

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，可以参考前面的查询表达式部分。

数组条件

可以通过数组方式批量设置查询条件，使用方式如下：

```
// 传入数组作为查询条件
Db::table('think_user')->where([
        ['name','=','thinkphp'],
    ['status','=',1]
])->select();
```

最后生成的SQL语句是

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

> 5.1的数组查询方式有所调整，是为了尽量避免数组方式的条件查询注入。

如果需要事先组装数组查询条件，可以使用：

```
$map[] = ['name','like','think'];
$map[] = ['status','=',1];
```

如果需要替换某些查询条件则可以使用

```
$map['status'] = ['status','=',1];
$map['name'] = ['name','like','think'];
$map['status'] = ['status','>=',1];
```

> 数组方式查询还有一些额外的复杂用法，我们会在后面的高级查询章节提及。

字符串条件

使用字符串条件直接查询和操作，例如：

```
Db::table('think_user')->where('type=1 AND status=1')->select();
```

最后生成的SQL语句是

```
SELECT * FROM think_user WHERE type=1 AND status=1
```

> 注意使用字符串查询条件和表达式查询的一个区别在于，不会对查询字段进行避免关键词冲突处理。

使用字符串条件的时候，建议配合预处理机制，确保更加安全，例如：

```
Db::table('think_user')
->where("id=:id and username=:name", ['id' => [1, \PDO::PARAM_INT] , 'nam
e' => 'thinkphp'])
->select();
```

或者使用

```
Db::table('think_user')
->where("id=:id and username=:name")
->bind(['id' => [1, \PDO::PARAM_INT] , 'name' => 'thinkphp'])
```

```
->select();
```

table

`table`方法主要用于指定操作的数据表。

用法

一般情况下，操作模型的时候系统能够自动识别当前对应的数据表，所以，使用table方法的情况通常是为了：

- 切换操作的数据表；
- 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`name`方式简化数据表前缀的传入，例如：

```
Db::name('user')->where('status>1')->select();
```

会自动获取当前模型对应的数据表前缀来生成`think_user`数据表名称。

需要注意的是`table`方法不会改变数据库的连接，所以你要确保当前连接的用户有权限操作相应的数据库和数据表。

如果需要对多表进行操作，可以这样使用：

```
Db::field('user.name,role.title')
->table('think_user user,think_role role')
->limit(10)->select();
```

为了尽量避免和mysql的关键字冲突，可以建议使用数组方式定义，例如：

```
Db::field('user.name,role.title')
->table(['think_user'=>'user','think_role'=>'role'])
->limit(10)->select();
```

使用数组方式定义的优势是可以避免因为表名和关键字冲突而出错的情况。

alias

`alias`用于设置当前数据表的别名，便于使用其他的连贯操作例如join方法等。

示例：

```
Db::table('think_user')
->alias('a')
->join('think_dept b ','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
```

可以传入数组批量设置数据表以及别名，例如：

```
Db::table('think_user')
->alias(['think_user'=>'user','think_dept'=>'dept'])
->join('think_dept','dept.user_id= user.id')
->select();
```

最终生成的SQL语句类似于：

```
SELECT * FROM think_user user INNER JOIN think_dept dept ON dept.user_id=
 user.id
```

field

`field`方法主要作用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 think_user
```

可以给某个字段设置别名，例如：

```
Db::table('think_user')->field('id,nickname as name')->select();
```

执行的SQL语句相当于：

```
SELECT id,nickname as name FROM think_user
```

使用SQL函数

可以在field方法中直接使用函数，例如：

```
Db::table('think_user')->field('id,SUM(score)')->select();
```

执行的SQL相当于：

```
SELECT id,SUM(score) FROM think_user
```

> 除了select方法之外，所有的查询方法，包括find等都可以使用field方法。

使用数组参数

field方法的参数可以支持数组，例如：

```
Db::table('think_user')->field(['id','title','content'])->select();
```

最终执行的SQL和前面用字符串方式是等效的。

数组方式的定义可以为某些字段定义别名，例如：

```
Db::table('think_user')->field(['id','nickname'=>'name'])->select();
```

执行的SQL相当于：

```
SELECT id,nickname as name FROM think_user
```

对于一些更复杂的字段要求，数组的优势则更加明显，例如：

```
Db::table('think_user')->field(['id','concat(name,"-",id)'=>'truename','L
EFT(title,7)'=>'sub_title'])->select();
```

执行的SQL相当于：

```
SELECT id,concat(name,'-',id) as truename,LEFT(title,7) as sub_title FROM
 think_user
```

> 对于带有复杂SQL函数的字段需求必须使用数组方式

获取所有字段

如果有一个表有非常多的字段，需要获取所有的字段（这个也许很简单，因为不调用field方法或者直接使用空的field方法都能做到）：

```
Db::table('think_user')->select();
Db::table('think_user')->field('*')->select();
```

上面的用法是等效的，都相当于执行SQL：

```
SELECT * FROM think_user
```

但是这并不是我说的获取所有字段，而是显式的调用所有字段（对于对性能要求比较高的系统，这个要求并不过分，起码是一个比较好的习惯），下面的用法可以完成预期的作用：

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

`field(true)`的用法会显式的获取数据表的所有字段列表，哪怕你的数据表有100个字段。

字段排除

如果我希望获取排除数据表中的`content`字段（文本字段的值非常耗内存）之外的所有字段值，我们就可以使用field方法的排除功能，例如下面的方式就可以实现所说的功能：

```
Db::table('think_user')->field('content',true)->select();
```

则表示获取除了content之外的所有字段，要排除更多的字段也可以：

```
Db::table('think_user')->field('user_id,content',true)->select();
//或者用
Db::table('think_user')->field(['user_id','content'],true)->select();
```

> 注意的是 字段排除功能不支持跨表和join操作。

用于写入

除了查询操作之外，`field`方法还有一个非常重要的安全功能--字段合法性检测。`field`方法结合数据库的写入方法使用就可以完成表单提交的字段合法性检测，如果我们在表单提交的处理方法中使用了：

```
Db::table('think_user')->field('title,email,content')->insert($data);
```

即表示表单中的合法字段只有`title`，`email`和`content`字段，无论用户通过什么手段更改或者添加了浏览器的提交字段，都会直接屏蔽。因为，其他所有字段我们都不希望由用户提交来决定，你可以通过自动完成功能定义额外需要自动写入的字段。

> 在开启数据表字段严格检查的情况下，提交了非法字段会抛出异常，可以在数据库设置文件中设置：

```
// 关闭严格字段检查
'fields_strict' =>      false,
```

strict

`strict` 方法用于设置是否严格检查字段名，用法如下：

```
// 关闭字段严格检查
Db::name('user')
    ->strict(false)
    ->insert($data);
```

注意，系统默认值是由数据库配置参数 `fields_strict` 决定，因此修改数据库配置参数可以进行全局的严格检查配置，如下：

```
// 关闭严格检查字段是否存在
'fields_strict'  => false,
```

> 如果开启字段严格检查的话，在更新和写入数据库的时候，一旦存在非数据表字段的值，则会抛出异常。

limit

`limit`方法主要用于指定查询和操作的数量。

> `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）。

对于大数据表，尽量使用`limit`限制查询结果，否则会导致很大的内存开销和性能问题。

page

page方法主要用于分页查询。

我们在前面已经了解了关于limit方法用于分页查询的情况，而page方法则是更人性化的进行分页查询的方法，例如还是以文章列表分页为例来说，如果使用limit方法，我们要查询第一页和第二页（假设我们每页输出10条数据）写法如下：

```
// 查询第一页数据
Db::table('think_article')->limit(0,10)->select();
// 查询第二页数据
Db::table('think_article')->limit(10,10)->select();
```

虽然利用扩展类库中的分页类Page可以自动计算出每个分页的limit参数，但是如果要自己写就比较费力了，如果用page方法来写则简单多了，例如：

```
// 查询第一页数据
Db::table('think_article')->page(1,10)->select();
// 查询第二页数据
Db::table('think_article')->page(2,10)->select();
```

显而易见的是，使用page方法你不需要计算每个分页数据的起始位置，page方法内部会自动计算。

`page` 方法还可以和limit方法配合使用，例如：

```
Db::table('think_article')->limit(25)->page(3)->select();
```

当page方法只有一个值传入的时候，表示第几页，而 `limit` 方法则用于设置每页显示的数量，也就是说上面的写法等同于：

```
Db::table('think_article')->page(3,25)->select();
```

order

`order` 方法用于对操作的结果排序或者优先级限制。

用法如下：

```
Db::table('think_user')
->where('status', 1)
->order('id', 'desc')
->limit(5)
->select();
```

```
 SELECT * FROM `think_user` WHERE `status` = 1 ORDER BY `id` desc LIMIT 5
```

> 如果没有指定 desc 或者 asc 排序规则的话，默认为 asc 。

支持使用数组对多个字段的排序，例如：

```
Db::table('think_user')
->where('status', 1)
->order(['order','id'=>'desc'])
->limit(5)
->select();
```

最终的查询SQL可能是

```
SELECT * FROM `think_user` WHERE `status` = 1 ORDER BY `order`,`id` desc
LIMIT 5
```

对于更新数据或者删除数据的时候可以用于优先级限制

```
Db::table('think_user')
->where('status', 1)
->order('id', 'desc')
->limit(5)
->delete();
```

生成的SQL

```
DELETE FROM `think_user` WHERE `status` = 1 ORDER BY `id` desc LIMIT 5
```

group

`GROUP` 方法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 us
er_id,test_time
```

having

`HAVING` 方法用于配合group方法完成从分组的结果中筛选（通常是聚合条件）数据。

`having` 方法只有一个参数，并且只能使用字符串，例如：

```
Db::table('think_user')
    ->field('username,max(score)')
    ->group('user_id')
    ->having('count(test_time)>3')
    ->select();
```

生成的SQL语句是：

```
SELECT username,max(score) FROM think_score GROUP BY user_id HAVING count
(test_time)>3
```

join

`JOIN` 方法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。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

```
join ( mixed join [, mixed $condition = null [, string $type = 'INNER']]
)
leftJoin ( mixed join [, mixed $condition = null ] )
rightJoin ( mixed join [, mixed $condition = null ] )
fullJoin ( mixed join [, mixed $condition = null ] )
```

参数

join

要关联的（完整）表名以及别名

支持的写法：

- 写法1：['完整表名或者子查询'=>'别名']
- 写法2：'不带数据表前缀的表名'（自动作为别名）
- 写法2：'不带数据表前缀的表名 别名'

condition

关联条件。可以为字符串或数组， 为数组时每一个元素都是一个关联条件。

type

关联类型。可以为:`INNER`、`LEFT`、`RIGHT`、`FULL`，不区分大小写，默认为`INNER`。

返回值

模型对象

举例

```
Db::table('think_artist')
->alias('a')
->join('work w','a.id = w.artist_id')
->join('card c','a.card_id = c.id')
->select();
```

```
Db::table('think_user')
->alias('a')
->join(['think_work'=>'w'],'a.id=w.artist_id')
->join(['think_card'=>'c'],'a.card_id=c.id')
->select();
```

默认采用INNER JOIN 方式，如果需要用其他的JOIN方式，可以改成

```
Db::table('think_user')
->alias('a')
->leftJoin('word w','a.id = w.artist_id')
->select();
```

表名也可以是一个子查询

```
$subsql = Db::table('think_work')
->where('status',1)
->field('artist_id,count(id) count')
->group('artist_id')
->buildSql();

Db::table('think_user')
->alias('a')
->join([$subsql=> 'w'], 'a.artist_id = w.artist_id')
->select();
```

union

UNION操作用于合并两个或多个 SELECT 语句的结果集。

使用示例：

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

闭包用法：

```
Db::field('name')
    ->table('think_user_0')
    ->union(function ($query) {
        $query->field('name')->table('think_user_1');
    })
    ->union(function ($query) {
        $query->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')
    ->unionAll('SELECT name FROM think_user_1')
    ->unionAll('SELECT name FROM think_user_2')
    ->select();
```

或者

```
Db::field('name')
    ->table('think_user_0')
    ->union(['SELECT name FROM think_user_1', 'SELECT name FROM think_use
r_2'], true)
    ->select();
```

每个union方法相当于一个独立的SELECT语句。

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

distinct

DISTINCT 方法用于返回唯一不同的值 。

例如数据库表中有以下数据

Modify	id	user_login	user_pass	user_name	create_time	status
编辑	1	chunice	89f0b495890138511edbca8d446aa63e	chunice	1463709258	1
编辑	2	admin	89f0b495890138511edbca8d446aa63e	admin	1463709258	1
编辑	3	admin	89f0b495890138511edbca8d446aa63e	admin	1463709258	1

以下代码会返回 `user_login` 字段不同的数据

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

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

返回以下数组

```
array(2) {
  [0] => array(1) {
    ["user_login"] => string(7) "chunice"
  }
  [1] => array(1) {
    ["user_login"] => string(5) "admin"
  }
}
```

`distinct` 方法的参数是一个布尔值。

lock

`Lock` 方法是用于数据库的锁机制，如果在查询或者执行操作的时候使用：

```
Db::name('user')->where('id',1)->lock(true)->find();
```

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）。

lock方法支持传入字符串用于一些特殊的锁定要求，例如：

```
Db::name('user')->where('id',1)->lock('lock in share mode')->find();
```

cache

`cache`方法用于查询缓存操作，也是连贯操作方法之一。

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

下面举例说明，例如，我们对find方法使用cache方法如下：

```
Db::table('think_user')->where('id',5)->cache(true)->find();
```

第一次查询结果会被缓存，第二次查询相同的数据的时候就会直接返回缓存中的内容，而不需要再次进行数据库查询操作。

默认情况下， 缓存有效期是由默认的缓存配置参数决定的，但`cache`方法可以单独指定，例如：

```
Db::table('think_user')->cache(true,60)->find();
// 或者使用下面的方式 是等效的
Db::table('think_user')->cache(60)->find();
```

表示对查询结果的缓存有效期60秒。

cache方法可以指定缓存标识：

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

> 指定查询缓存的标识可以使得查询缓存更有效率。

这样，在外部就可以通过`\think\Cache`类直接获取查询缓存的数据，例如：

```
$result = Db::table('think_user')->cache('key',60)->find();
$data = \think\Cache::get('key');
```

`cache`方法支持设置缓存标签，例如：

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

缓存自动更新

这里的缓存自动更新是指一旦数据更新或者删除后会自动清理缓存（下次获取的时候会自动重新缓存）。

当你删除或者更新数据的时候，可以调用相同`key`的`cache`方法，会自动更新（清除）缓存，例如：

```
Db::table('think_user')->cache('user_data')->select([1,3,5]);
Db::table('think_user')->cache('user_data')->update(['id'=>1,'name'=>'thi
nkphp']);
Db::table('think_user')->cache('user_data')->select([1,5]);
```

最后查询的数据不会受第一条查询缓存的影响，确保查询和更新或者删除使用相同的缓存标识才能自动清除缓存。

如果使用`find`方法并且使用主键查询的情况，不需要指定缓存标识，会自动清理缓存，例如：

```
Db::table('think_user')->cache(true)->find(1);
Db::table('think_user')->update(['id'=>1,'name'=>'thinkphp']);
Db::table('think_user')->cache(true)->find(1);
```

最后查询的数据会是更新后的数据。

comment

COMMENT方法 用于在生成的SQL语句中添加注释内容，例如：

```
Db::table('think_score')->comment('查询考试前十名分数')
    ->field('username,score')
    ->limit(10)
    ->order('score desc')
    ->select();
```

最终生成的SQL语句是：

```
SELECT username,score FROM think_score ORDER BY score desc LIMIT 10 /* 查
询考试前十名分数 */
```

fetchSql

`fetchSql`用于直接返回SQL而不是执行查询，适用于任何的CURD操作方法。 例如：

```
echo Db::table('think_user')->fetchSql(true)->find(1);
```

输出结果为：

```
SELECT * FROM think_user where `id` = 1
```

force

force 方法用于数据集的强制索引操作，例如：

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

对查询强制使用`user`索引，`user`必须是数据表实际创建的索引名称。

partition

`partition` 方法用于是数据库水平分表

```
partition($data, $field, $rule);
// $data 分表字段的数据
// $field 分表字段的名称
// $rule 分表规则
```

注意：不要使用任何 SQL 语句中会出现的关键字当表名、字段名，例如 order 等。会导致数据模型拼装 SQL 语句语法错误。

`partition` 方法用法如下：

```
// 用于写入
$data = [
    'user_id'   => 110,
    'user_name' => 'think'
];

$rule = [
    'type' => 'mod', // 分表方式
    'num'  => 10     // 分表数量
];

Db::name('log')
    ->partition(['user_id' => 110], "user_id", $rule)
    ->insert($data);

// 用于查询
Db::name('log')
    ->partition(['user_id' => 110], "user_id", $rule)
    ->where(['user_id' => 110])
    ->select();
```

failException

`failException` 设置查询数据为空时是否需要抛出异常，如果不传入任何参数，默认为开启，用于 `select` 和 `find` 方法，例如：

```
// 数据不存在的话直接抛出异常
Db::name('blog')
        ->where('status',1)
    ->failException()
    ->select();
// 数据不存在返回空数组 不抛异常
Db::name('blog')
        ->where('status',1)
    ->failException(false)
    ->select();
```

或者可以使用更方便的查空报错

```
// 查询多条
Db::name('blog')
        ->where('status', 1)
    ->selectOrFail();

// 查询单条
Db::name('blog')
        ->where('status', 1)
    ->findOrFail();
```

sequence

`sequence`方法用于`pgsql`数据库指定自增序列名，其它数据库不必使用，用法为：

```
Db::name('user')
->sequence('user_id_seq')
->insert(['name'=>'thinkphp']);
```

聚合查询

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

方法	说明
count	统计数量，参数是要统计的字段名（可选）
max	获取最大值，参数是要统计的字段名（必须）
min	获取最小值，参数是要统计的字段名（必须）
avg	获取平均值，参数是要统计的字段名（必须）
sum	获取总分，参数是要统计的字段名（必须）

> 聚合方法如果没有数据，默认都是0，聚合查询都可以配合其它查询条件

> `V5.1.5+` 版本开始，聚合查询可以支持 `JSON` 字段类型

用法示例

获取用户数：

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

实际生成的SQL语句是：

```
SELECT COUNT(*) AS tp_count FROM `think_user` LIMIT 1
```

或者根据字段统计：

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

生成的SQL语句是：

```
SELECT COUNT(id) AS tp_count FROM `think_user` LIMIT 1
```

获取用户的最大积分：

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

生成的SQL语句是：

```
SELECT MAX(score) AS tp_max FROM `think_user` LIMIT 1
```

如果你要获取的最大值不是一个数值，可以使用第二个参数关闭强制转换

```
Db::table('think_user')->max('name',false);
```

获取积分大于0的用户的最小积分：

```
Db::table('think_user')->where('score', '>', 0)->min('score');
```

和max方法一样，min也支持第二个参数用法

```
Db::table('think_user')->where('score', '>', 0)->min('name',false);
```

获取用户的平均积分：

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

生成的SQL语句是：

```
SELECT AVG(score) AS tp_avg FROM `think_user` LIMIT 1
```

统计用户的总成绩：

```
Db::table('think_user')->where('id',10)->sum('score');
```

生成的SQL语句是：

```
SELECT SUM(score) AS tp_sum FROM `think_user` LIMIT 1
```

时间查询

时间比较

使用 `where` 方法

`where` 方法支持时间比较，例如：

```
// 大于某个时间
where('create_time', '> time', '2016-1-1');
// 小于某个时间
where('create_time', '<= time', '2016-1-1');
// 时间区间查询
where('create_time', 'between time', ['2015-1-1', '2016-1-1']);
```

第三个参数可以传入任何有效的时间表达式，会自动识别你的时间字段类型，支持的时间类型包括 `timestamps`、`datetime`、`date` 和 `int`。

使用 `whereTime` 方法

`whereTime` 方法提供了日期和时间字段的快捷查询，示例如下：

```
// 大于某个时间
Db::name('user')
    ->whereTime('birthday', '>=', '1970-10-1')
    ->select();
// 小于某个时间
Db::name('user')
    ->whereTime('birthday', '<', '2000-10-1')
    ->select();
// 时间区间查询
Db::name('user')
    ->whereTime('birthday', 'between', ['1970-10-1', '2000-10-1'])
    ->select();
// 不在某个时间区间
Db::name('user')
    ->whereTime('birthday', 'not between', ['1970-10-1', '2000-10-1'])
    ->select();
```

针对时间的区间查询，系统还提供了一个 `whereBetweenTime` 快速方法

```
// 查询2017年上半年注册的用户
Db::name('user')
    ->whereBetweenTime('create_time', '2017-01-01', '2017-06-30')
    ->select();
// 查询2017年6月1日注册的用户
```

```
Db::name('user')
    ->whereBetweenTime('create_time', '2017-06-01')
    ->select();
```

没有指定结束时间的话，表示查询当天。

时间表达式

还提供了更方便的时间表达式查询，例如：

```
// 获取今天的博客
Db::name('blog')
    ->whereTime('create_time', 'today')
    ->select();

// 获取昨天的博客
Db::name('blog')
    ->whereTime('create_time', 'yesterday')
    ->select();

// 获取本周的博客
Db::name('blog')
    ->whereTime('create_time', 'week')
    ->select();

// 获取上周的博客
Db::name('blog')
    ->whereTime('create_time', 'last week')
    ->select();

// 获取本月的博客
Db::name('blog')
    ->whereTime('create_time', 'month')
    ->select();

// 获取上月的博客
Db::name('blog')
    ->whereTime('create_time', 'last month')
    ->select();

// 获取今年的博客
Db::name('blog')
    ->whereTime('create_time', 'year')
    ->select();

// 获取去年的博客
Db::name('blog')
    ->whereTime('create_time', 'last year')
    ->select();
```

如果查询当天、本周、本月和今年的时间，还可以简化为：

```
// 获取今天的博客
Db::name('blog')
    ->whereTime('create_time', 'd')
    ->select();

// 获取本周的博客
Db::name('blog')
    ->whereTime('create_time', 'w')
    ->select();

// 获取本月的博客
Db::name('blog')
    ->whereTime('create_time', 'm')
    ->select();

// 获取今年的博客
Db::name('blog')
    ->whereTime('create_time', 'y')
    ->select();
```

还可以使用下面的时间表达式进行时间查询

```
// 查询两个小时内的博客
Db::name('blog')
        ->whereTime('create_time','-2 hours')
    ->select();
```

高级查询

快捷查询

快捷查询方式是一种多字段相同查询条件的简化写法，可以进一步简化查询条件的写法，在多个字段之间用 `|` 分割表示 OR 查询，用 `&` 分割表示 AND 查询，可以实现下面的查询，例如：

```
Db::table('think_user')
    ->where('name|title','like','thinkphp%')
    ->where('create_time&update_time','>',0)
    ->find();
```

生成的查询SQL是：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' OR `title` LIK
E 'thinkphp%' ) AND ( `create_time` > 0 AND `update_time` > 0 ) LIMIT 1
```

> 快捷查询支持所有的查询表达式。

区间查询

区间查询是一种同一字段多个查询条件的简化写法，例如：

```
Db::table('think_user')
    ->where('name', ['like', '%thinkphp%'], ['like', '%kancloud%'], 'or')
    ->where('id', ['>', 0], ['<>', 10], 'and')
    ->find();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE '%thinkphp%' OR `name` LIK
E '%kancloud%' ) AND ( `id` > 0 AND `id` <> 10 ) LIMIT 1
```

> 区间查询的查询条件必须使用数组定义方式，支持所有的查询表达式。

下面的查询方式是错误的：

```
Db::table('think_user')
    ->where('name', ['like', 'thinkphp%'], ['like', '%thinkphp'])
    ->where('id', 5, ['<>', 10], 'or')
    ->find();
```

区间查询其实可以用下面的方式替代，更容易理解，因为查询构造器支持对同一个字段多次调用查询条件，例如：

```
Db::table('think_user')
    ->where('name', 'like', '%think%')
    ->where('name', 'like', '%php%')
    ->where('id', 'in', [1, 5, 80, 50])
    ->where('id', '>', 10)
    ->find();
```

批量（字段）查询

可以进行多个条件的批量条件查询定义，例如：

```
Db::table('think_user')
    ->where([
        ['name', 'like', 'thinkphp%'],
        ['title', 'like', '%thinkphp'],
        ['id', '>', 0],
        ['status', '=', 1],
    ])
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE `name` LIKE 'thinkphp%' AND `title` LIKE
 '%thinkphp' AND `id` > 0 AND `status` = '1'
```

数组查询方式，确保你的查询数组不能被用户提交数据控制，用户提交的表单数据应该是作为查询数组的一个元素传入，如下：

```
Db::table('think_user')
    ->where([
        ['name', 'like', $name . '%'],
        ['title', 'like', '%' . $title],
        ['id', '>', $id],
        ['status', '=', $status],
    ])
    ->select();
```

> 注意，相同的字段的多次查询条件可能会合并，如果希望某一个 where 方法里面的条件单独处理，可以使用下面的方式，避免被其它条件影响。

```
$map = [
        ['name', 'like', 'thinkphp%'],
        ['title', 'like', '%thinkphp'],
        ['id', '>', 0],
    ];
Db::table('think_user')
    ->where([ $map ])
    ->where('status',1)
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' AND `title` LI
KE '%thinkphp' AND `id` > 0 ) AND `status` = '1'
```

如果使用下面的多个条件组合

```
$map1 = [
        ['name', 'like', 'thinkphp%'],
        ['title', 'like', '%thinkphp'],
    ];

$map2 = [
        ['name', 'like', 'kancloud%'],
        ['title', 'like', '%kancloud'],
    ];

Db::table('think_user')
    ->whereOr([ $map1, $map2 ])
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' AND `title` LI
KE '%thinkphp' ) OR ( `name` LIKE 'kancloud%' AND `title` LIKE '%kancloud
' )
```

> 善用多维数组查询，可以很方便的拼装出各种复杂的SQL语句

闭包查询

```
$name = 'thinkphp';
$id = 10;
Db::table('think_user')->where(function ($query) use($name, $id) {
    $query->where('name', $name)
        ->whereOr('id', '>', $id);
})->select();
```

生成的SQL语句为：

```
SELECT * FROM `think_user` WHERE ( `name` = 'thinkphp' OR `id` > 10 )
```

> 可见每个闭包条件两边也会自动加上括号

混合查询

可以结合前面提到的所有方式进行混合查询，例如：

```
Db::table('think_user')
    ->where('name', ['like', 'thinkphp%'], ['like', '%thinkphp'])
    ->where(function ($query) {
        $query->where('id', ['<', 10], ['>', 100], 'or');
    })
    ->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' AND `name` LIK
E '%thinkphp' ) AND ( `id` < 10 or `id` > 100 )
```

字符串条件查询

对于一些实在复杂的查询，也可以直接使用原生SQL语句进行查询，例如：

```
Db::table('think_user')
    ->where('id > 0 AND name LIKE "thinkphp%"')
    ->select();
```

为了安全起见，我们可以对字符串查询条件使用参数绑定，例如：

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

```
    ->where('id > :id AND name LIKE :name ', ['id' => 0, 'name' => 'think
php%'])
    ->select();
```

使用Query对象查询（ V5.1.5+ ）

V5.1.5+ 版本开始，可以通过调用一次 where 方法传入 Query 对象来进行查询。

```
$query = new \think\db\Query;
$query->where('id','>',0)
        ->where('name','like','%thinkphp');

Db::table('think_user')
    ->where($query)
    ->select();
```

> Query对象的 where 方法仅能调用一次，如果 query 对象里面使用了非常查询条件的链式方法，则不会传入当前查询。

```
$query = new \think\db\Query;
$query->where('id','>',0)
        ->where('name','like','%thinkphp')
    ->order('id','desc') // 不会传入后面的查询
    ->field('name,id'); // 不会传入后面的查询

Db::table('think_user')
    ->where($query)
    ->where('title','like','thinkphp%') // 有效
    ->select();
```

快捷方法

系统封装了一系列快捷方法，用于简化查询，包括：

方法	作用
whereOr	字段OR查询
whereXor	字段XOR查询
whereNull	查询字段是否为Null
whereNotNull	查询字段是否不为Null
whereIn	字段IN查询
whereNotIn	字段NOT IN查询
whereBetween	字段BETWEEN查询
whereNotBetween	字段NOT BETWEEN查询

whereLike	字段LIKE查询
whereNotLike	字段NOT LIKE查询
whereExists	EXISTS条件查询
whereNotExists	NOT EXISTS条件查询
whereExp	表达式查询
whereColumn	比较两个字段

下面举例说明下两个字段比较的查询条件 whereColumn 方法的用法。

查询 update_time 大于 create_time 的用户数据

```
Db::table('think_user')
    ->whereColumn('update_time','>','create_time')
    ->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE ( `update_time` > create_time )
```

查询 name 和 nickname 相同的用户数据

```
Db::table('think_user')
    ->whereColumn('name','=','nickname')
    ->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE ( `name` = nickname )
```

相同字段条件也可以简化为

```
Db::table('think_user')
    ->whereColumn('name','nickname')
    ->select();
```

动态查询

查询构造器还提供了两个动态查询机制，用于简化查询条件，包括 getBy 和 getFieldBy 。

动态查询	描述
`whereFieldName`	查询某个字段的值
`whereOrFieldName`	查询某个字段的值
`getByFieldName`	根据某个字段查询
`getFieldByFieldName`	根据某个字段获取某个值

其中`FieldName`表示数据表的实际字段名称的驼峰法表示，假设数据表`user`中有`email`和`nick_name`字段，我们可以这样来查询。

```
// 根据邮箱（email）查询用户信息
$user = Db::table('user')
        ->whereEmail('thinkphp@qq.com')
    ->find();

// 根据昵称（nick_name）查询用户
$email = Db::table('user')
    ->whereNickName('like', '%流年%')
    ->select();

// 根据邮箱查询用户信息
$user = Db::table('user')
    ->getByEmail('thinkphp@qq.com');

// 根据昵称（nick_name）查询用户信息
$user = Db::table('user')
    ->field('id,name,nick_name,email')
    ->getByNickName('流年');

// 根据邮箱查询用户的昵称
$nickname = Db::table('user')
    ->getFieldByEmail('thinkphp@qq.com', 'nick_name');

// 根据昵称（nick_name）查询用户邮箱
$email = Db::table('user')
    ->getFieldByNickName('流年', 'email');
```

> `getBy`和`getFieldBy`方法只会查询一条记录，可以和其它的链式方法搭配使用

条件查询

5.1开始查询构造器支持条件查询，例如：

```
Db::name('user')->when($condition, function ($query) {
    // 满足条件后执行
    $query->where('score', '>', 80)->limit(10);
})->select();
```

并且支持不满足条件的分支查询

```
Db::name('user')->when($condition, function ($query) {
    // 满足条件后执行
    $query->where('score', '>', 80)->limit(10);
}, function ($query) {
    // 不满足条件执行
    $query->where('score', '>', 60);
});
```

视图查询

视图查询可以实现不依赖数据库视图的多表查询，并不需要数据库支持视图，是JOIN方法的推荐替代方法，例如：

```
Db::view('User', 'id,name')
    ->view('Profile', 'truename,phone,email', 'Profile.user_id=User.id')
    ->view('Score', 'score', 'Score.user_id=Profile.id')
    ->where('score', '>', 80)
    ->select();
```

生成的SQL语句类似于：

```
SELECT User.id,User.name,Profile.truename,Profile.phone,Profile.email,Sco
re.score FROM think_user User INNER JOIN think_profile Profile ON Profile
.user_id=User.id INNER JOIN think_socre Score ON Score.user_id=Profile.id
 WHERE Score.score > 80
```

> 注意，视图查询无需调用 `table` 和 `join` 方法，并且在调用 `where` 和 `order` 方法的时候只需要使用字段名而不需要加表名。

默认使用`INNER join`查询，如果需要更改，可以使用：

```
Db::view('User', 'id,name')
    ->view('Profile', 'truename,phone,email', 'Profile.user_id=User.id',
'LEFT')
    ->view('Score', 'score', 'Score.user_id=Profile.id', 'RIGHT')
    ->where('score', '>', 80)
    ->select();
```

生成的SQL语句类似于：

```
SELECT User.id,User.name,Profile.truename,Profile.phone,Profile.email,Sco
re.score FROM think_user User LEFT JOIN think_profile Profile ON Profile.
user_id=User.id RIGHT JOIN think_socre Score ON Score.user_id=Profile.id
WHERE Score.score > 80
```

可以使用别名：

```
Db::view('User', ['id' => 'uid', 'name' => 'account'])
    ->view('Profile', 'truename,phone,email', 'Profile.user_id=User.id')
    ->view('Score', 'score', 'Score.user_id=Profile.id')
    ->where('score', '>', 80)
    ->select();
```

生成的SQL语句变成：

```
SELECT User.id AS uid,User.name AS account,Profile.truename,Profile.phone
,Profile.email,Score.score FROM think_user User INNER JOIN think_profile
Profile ON Profile.user_id=User.id INNER JOIN think_socre Score ON Score.
user_id=Profile.id WHERE Score.score > 80
```

可以使用数组的方式定义表名以及别名，例如：

```
Db::view(['think_user' => 'member'], ['id' => 'uid', 'name' => 'account']
)
    ->view('Profile', 'truename,phone,email', 'Profile.user_id=member.id'
)
    ->view('Score', 'score', 'Score.user_id=Profile.id')
    ->where('score', '>', 80)
    ->select();
```

生成的SQL语句变成：

```
SELECT member.id AS uid,member.name AS account,Profile.truename,Profile.p
hone,Profile.email,Score.score FROM think_user member INNER JOIN think_pr
ofile Profile ON Profile.user_id=member.id INNER JOIN think_socre Score O
N Score.user_id=Profile.id WHERE Score.score > 80
```

JSON字段

JSON字段

从`V5.1.4+`版本开始，强化了JSON字段的操作支持。

如果你的`user`表有一个`info`字段是`JSON`类型的（或者说你存储的是JSON格式，但并非是要JSON字段类型），你可以使用下面的方式操作数据。

JSON数据写入

```
$user['name'] = 'thinkphp';
$user['info'] = [
        'email'    => 'thinkphp@qq.com',
        'nickname' => '流年',
];
Db::name('user')
        ->json(['info'])
        ->insert($user);
```

JSON数据查询

查询整个JSON数据：

```
$user = Db::name('user')
        ->json(['info'])
        ->find(1);
dump($user);
```

查询条件为JSON数据

```
$user = Db::name('user')
        ->json(['info'])
    ->where('info->nickname','ThinkPHP')
        ->find();
dump($user);
```

JSON数据更新

完整JSON数据更新

```
$data['info'] = [
        'email'    => 'kancloud@qq.com',
        'nickname' => 'kancloud',
];
```

```
Db::name('user')
        ->json(['info'])
    ->where('id',1)
        ->update($data);
```

单个JSON数据更新

```
$data['info->nickname'] = 'ThinkPHP';
Db::name('user')
        ->json(['info'])
    ->where('id',1)
        ->update($data);
```

子查询

首先构造子查询SQL，可以使用下面三种的方式来构建子查询。

使用 `fetchSql` 方法

fetchSql方法表示不进行查询而只是返回构建的SQL语句，并且不仅仅支持`select`，而是支持所有的CURD查询。

```
$subQuery = Db::table('think_user')
    ->field('id,name')
    ->where('id', '>', 10)
    ->fetchSql(true)
    ->select();
```

生成的subQuery结果为：

```
SELECT `id`,`name` FROM `think_user` WHERE `id` > 10
```

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。

然后使用子查询构造新的查询：

```
Db::table($subQuery . ' a')
    ->where('a.name', 'like', 'thinkphp')
    ->order('id', 'desc')
    ->select();
```

生成的SQL语句为：

```
SELECT * FROM ( SELECT `id`,`name` FROM `think_user` WHERE `id` > 10 ) a
WHERE a.name LIKE 'thinkphp' ORDER BY `id` desc
```

使用闭包构造子查询

`IN/NOT IN` 和 `EXISTS/NOT EXISTS` 之类的查询可以直接使用闭包作为子查询，例如：

```
Db::table('think_user')
    ->where('id', 'IN', function ($query) {
        $query->table('think_profile')->where('status', 1)->field('id');
    })
    ->select();
```

生成的SQL语句是

```
SELECT * FROM `think_user` WHERE `id` IN ( SELECT `id` FROM `think_profil
e` WHERE `status` = 1 )
```

```
Db::table('think_user')
    ->where(function ($query) {
        $query->table('think_profile')->where('status', 1);
    }, 'exists')
    ->find();
```

生成的SQL语句为

```
SELECT * FROM `think_user` WHERE EXISTS ( SELECT * FROM `think_profile` W
HERE `status` = 1 )
```

> 除了上述查询条件外，比较运算也支持使用闭包子查询

原生查询

Db 类支持原生 SQL 查询操作，主要包括下面两个方法：

query 方法

query 方法用于执行 SQL 查询操作，如果数据非法或者查询错误则返回false，否则返回查询结果数据集（同 select 方法）。

使用示例：

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

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

execute 方法

execute 用于更新和写入数据的sql操作，如果数据非法或者查询错误则返回 false ，否则返回影响的记录数。

使用示例：

```
Db::execute("update think_user set name='thinkphp' where status=1");
```

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

参数绑定

支持在原生查询的时候使用参数绑定，包括问号占位符或者命名占位符，例如：

```
Db::query("select * from think_user where id=? AND status=?", [8, 1]);
// 命名绑定
Db::execute("update think_user set name=:name where status=:status", ['na
me' => 'thinkphp', 'status' => 1]);
```

> 注意不支持对表名使用参数绑定

查询事件

查询事件

数据库的CURD操作支持事件，包括：

事件	描述
before_select	`select` 查询前回调
before_find	`find` 查询前回调
after_insert	`insert` 操作成功后回调
after_update	`update` 操作成功后回调
after_delete	`delete` 操作成功后回调

查询事件仅支持 `find` 、 `select` 、 `insert` 、 `update` 和 `delete` 方法。

注册事件

使用下面的方法注册数据库查询事件

```
Db::event('after_insert', 'callback');
Db::event('before_select', function ($query) {
    // 事件处理
    return $result;
});
```

> 查询事件的方法参数只有一个：当前的查询对象。但你可以通过依赖注入的方式添加额外的参数。

事务操作

> 使用事务处理的话，需要数据库引擎支持事务处理。比如 MySQL 的 MyISAM 不支持事务处理，需要使用 InnoDB 引擎。

最简单的方式是使用 `transaction` 方法操作数据库事务，当闭包中的代码发生异常会自动回滚，例如：

```
Db::transaction(function () {
    Db::table('think_user')->find(1);
    Db::table('think_user')->delete(1);
});
```

也可以手动控制事务，例如：

```
// 启动事务
Db::startTrans();
try {
    Db::table('think_user')->find(1);
    Db::table('think_user')->delete(1);
    // 提交事务
    Db::commit();
} catch (\Exception $e) {
    // 回滚事务
    Db::rollback();
}
```

> 注意在事务操作的时候，确保你的数据库连接使用的是同一个。

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。一旦设置了SQL监听，则SQL日志需要自己接管。

存储过程

数据访问层支持存储过程调用，调用数据库存储过程使用下面的方法：

```
$resultSet = Db::query('call procedure_name');
foreach ($resultSet as $result) {

}
```

存储过程返回的是一个数据集，如果你的存储过程不需要返回任何的数据，那么也可以使用execute方法：

```
Db::execute('call procedure_name');
```

存储过程可以支持输入和输出参数，以及进行参数绑定操作。

```
$resultSet = Db::query('call procedure_name(:in_param1,:in_param2,:out_pa
ram)', [
    'in_param1' => $param1,
    'in_param2' => [$param2, PDO::PARAM_INT],
    'out_param' => [$outParam, PDO::PARAM_STR | PDO::PARAM_INPUT_OUTPUT,
4000],
]);
```

输出参数的绑定必须额外使用 `PDO::PARAM_INPUT_OUTPUT` ，并且可以和输入参数公用一个参数。

> 无论存储过程内部做了什么操作，每次存储过程调用仅仅被当成一次查询。

数据集

数据库的查询结果也就是数据集，默认的配置下，数据集的类型是一个二维数组，我们可以配置成数据集类，就可以支持对数据集更多的对象化操作，需要使用数据集类功能，可以配置数据库的 `resultset_type` 参数如下：

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

返回的数据集对象是 `think\Collection` ，提供了和数组无差别用法，并且另外封装了一些额外的方法。

> 在模型中进行数据集查询，全部返回数据集对象，但使用的是 `think\model\Collection` 类，但用法是一致的。

可以直接使用数组的方式操作数据集对象，例如：

```
// 获取数据集
$users = Db::name('user')->select();
// 直接操作第一个元素
$item  = $users[0];
// 获取数据集记录数
```

```
$count = count($users);
// 遍历数据集
foreach($users as $user){
    echo $user['name'];
    echo $user['id'];
}
```

需要注意的是，如果要判断数据集是否为空，不能直接使用`empty`判断，而必须使用数据集对象的`isEmpty`方法判断，例如：

```
$users = Db::name('user')->select();
if($users->isEmpty()){
    echo '数据集为空';
}
```

`Collection`类包含了下列主要方法：

方法	描述
isEmpty	是否为空
toArray	转换为数组
all	所有数据
merge	合并其它数据
diff	比较数组，返回差集
flip	交换数据中的键和值
intersect	比较数组，返回交集
keys	返回数据中的所有键名
pop	删除数据中的最后一个元素
shift	删除数据中的第一个元素
unshift	在数据开头插入一个元素
reduce	通过使用用户自定义函数，以字符串返回数组
reverse	数据倒序重排
chunk	数据分隔为多个数据块
each	给数据的每个元素执行回调
filter	用回调函数过滤数据中的元素
column	返回数据中的指定列
sort	对数据排序
shuffle	将数据打乱
slice	截取数据中的一部分

分布式数据库

分布式支持

数据访问层支持分布式数据库，包括读写分离，要启用分布式数据库，需要开启数据库配置文件中的 `deploy` 参数：

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

> 启用分布式数据库后， `hostname` 参数是关键， `hostname` 的个数决定了分布式数据库的数量，默认情况下第一个地址就是主服务器。

主从服务器支持设置不同的连接参数，包括：

连接参数
username
password
hostport
database
dsn
charset

如果主从服务器的上述参数一致的话，只需要设置一个，对于不同的参数，可以分别设置，例如：

```
return [
    // 启用分布式数据库
    'deploy'   => 1,
    // 数据库类型
    'type'     => 'mysql',
    // 服务器地址
    'hostname' => '192.168.1.1,192.168.1.2,192.168.1.3',
    // 数据库名
    'database' => 'demo',
    // 数据库用户名
    'username' => 'root,slave,slave',
    // 数据库密码
    'password' => '123456',
    // 数据库连接端口
    'hostport' => '',
    // 数据库字符集
    'charset'  => 'utf8',
];
```

> 记住，要么相同，要么每个都要设置。

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

```
    'rw_separate' => true,
```

在读写分离的情况下，默认第一个数据库配置是主服务器的配置信息，负责写入数据，如果设置了`master_num`参数，则可以支持多个主服务器写入（每次随机连接其中一个主服务器）。其它的地址都是从数据库，负责读取数据，数量不限制。每次连接从服务器并且进行读取操作的时候，系统会随机进行在从服务器中选择。同一个数据库连接的每次请求只会连接一次主服务器和从服务器，如果某次请求的从服务器连接不上，会自动切换到主服务器进行查询操作。

如果不希望随机读取，或者某种情况下其它从服务器暂时不可用，还可以设置`slave_no`指定固定服务器进行读操作，`slave_no`指定的序号表示`hostname`中数据库地址的序号，从`0`开始。

调用查询类或者模型的`CURD`操作的话，系统会自动判断当前执行的方法是读操作还是写操作并自动连接主从服务器，如果你用的是原生SQL，那么需要注意系统的默认规则： 写操作必须用数据库的`execute`方法，读操作必须用数据库的`query`方法，否则会发生主从读写错乱的情况。

发生下列情况的话，会自动连接主服务器：

- 使用了数据库的写操作方法（`execute`/`insert`/`update`/`delete`以及衍生方法）；
- 如果调用了数据库事务方法的话，会自动连接主服务器；
- 从服务器连接失败，会自动连接主服务器；
- 调用了查询构造器的`lock`方法；
- 调用了查询构造器的`master`方法

> 主从数据库的数据同步工作不在框架实现，需要数据库考虑自身的同步或者复制机制。如果在大数据量或者特殊的情况下写入数据后可能会存在同步延迟的情况，可以调用`master()`方法进行主库查询操作。

> 在实际生产环境中，很多云主机的数据库分布式实现机制和本地开发会有所区别，但通常会采下面用两种方式：
>
> - 第一种：提供了写IP和读IP（一般是虚拟IP），进行数据库的读写分离操作；
> - 第二种：始终保持同一个IP连接数据库，内部会进行读写分离IP调度（阿里云就是采用该方式）。

模型

定义

模型定义

定义一个`User`模型类很简单：

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

use think\Model;

class User extends Model
{
}
```

> 请确保你已经在数据库配置文件中配置了数据库连接信息，如不清楚请参考数据库一章

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。

> 模型自动对应的数据表名称都是遵循小写+下划线规范，如果你的表名有大写的情况，必须通过设置模型的table属性。

如果担心模型的名称和PHP关键字冲突，可以启用类后缀功能，只需要在应用配置文件`app.php`中设置：

```
    // 开启应用类库后缀
    'class_suffix'           => true,
```

> 开启后，所有的应用类库定义的时候都需要加上对应后缀，包括控制器类。

这样`app\index\model\User`类定义就要改成

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

use think\Model;

class UserModel extends Model
{
}
```

并且文件名也要改为`UserModel.php`。

> 大多数情况下，不同模块的模型是不需要独立的，因此可以统一在 common 模块下面定义模型。

模型设置

默认主键为`id`，如果你没有使用`id`作为主键名，需要在模型中设置属性：

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

use think\Model;

class User extends Model
{
    protected $pk = 'uid';
}
```

> 5.1中模型不会自动获取主键名称，必须设置pk属性。

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;

use think\Model;

class User extends Model
{
    // 设置当前模型对应的完整数据表名称
    protected $table = 'think_user';

    // 设置当前模型的数据库连接
```

```
    protected $connection = 'db_config';
}
```

`connection`属性的建议用配置参数名（需要在`database.php`中添加）而不是具体的连接信息，从而避免把数据库连接固化在代码里面。

常用的模型设置属性包括（以下属性都不是必须设置）：

属性	描述
name	模型名（默认为当前不含后缀的模型类名）
table	数据表名（默认自动获取）
pk	主键名（默认为`id`）
connection	数据库连接（默认读取数据库配置）
query	模型使用的查询类名称
field	模型对应数据表的字段列表（数组）

模型初始化

模型同样支持初始化，与控制器的初始化不同的是，模型的初始化是定义`Model`的`init`方法，具体如下

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

use think\Model;

class User extends Model
{

    // 模型初始化
    protected static function init()
    {
        //TODO:初始化内容
    }
}
```

模型初始化方法通常用于注册模型的事件操作。

> `init`必须是静态方法，并且只在第一次实例化的时候执行

模型操作

> 在模型中除了可以调用数据库类的方法之外（换句话说，数据库的所有查询方法模型中都

> 可以支持），可以定义自己的方法，所以也可以把模型看成是数据库的增强版。

模型的查询方法无需和数据库查询一样调用`table`或者`name`方法，因为模型会按照规则自动匹配对应的数据表，例如：

```
Db::name('user')->where('id','>',10)->select();
```

改成模型操作的话就变成

```
User::where('id','>',10)->select();
```

虽然看起来是相同的查询条件，但一个最明显的区别是查询结果的类型不同。第一种方式的查询结果是一个（二维）数组，而第二种方式的查询结果是包含了模型（集合）的数据集。不过，在大多数情况下，这二种返回类型的使用方式并无明显区别。

模型操作和数据库操作的另外一个显著区别是模型支持包括获取器、修改器、自动完成在内的一系列自动化操作和事件，简化了数据的存取操作，但随之而来的是性能有所下降（其实并没下降，而是自动帮你处理了一些原本需要手动处理的操作），后面会逐步领略到模型的这些特色功能。

新增

模型数据的新增和数据库的新增数据有所区别，数据库的新增只是单纯的写入给定的数据，而模型的数据写入会包含修改器、自动完成以及模型事件等环节，数据库的数据写入参考数据库章节。

添加一条数据

第一种是实例化模型对象后赋值并保存：

```
$user           = new User;
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
```

> `save`方法返回影响的记录数，并且只有当`before_insert`事件返回`false`的时候返回`false`

也可以直接传入数据到`save`方法批量赋值：

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

或者直接在实例化的时候传入数据

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

> 实例化传入的模型数据也不会经过修改器处理。

如果需要过滤非数据表字段的数据，可以使用：

```
$user = new User;
// 过滤post数组中的非数据表字段数据
$user->allowField(true)->save($_POST);
```

如果你通过外部提交赋值给模型，并且希望指定某些字段写入，可以使用：

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

最佳的建议是模型数据赋值之前就进行数据过滤，例如：

```
$user = new User;
// 过滤post数组中的非数据表字段数据
$data = Request::only(['name','email']);
$user->save($data);
```

> save 方法新增数据返回的是写入的记录数（通常是 1 ），而不是自增主键值。

获取自增ID

如果要获取新增数据的自增ID，可以使用下面的方式：

```
$user           = new User;
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
// 获取自增ID
echo $user->id;
```

这里其实是获取模型的主键，如果你的主键不是`id`，而是`user_id`的话，其实获取自增ID就变成这样：

```
$user           = new User;
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
// 获取自增ID
echo $user->user_id;
```

> 不要在同一个实例里面多次新增数据，如果确实需要多次新增，可以使用后面的静态方法

> 处理。

添加多条数据

支持批量新增，可以使用：

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

> saveAll方法新增数据返回的是包含新增模型（带自增ID）的数据集对象。

`saveAll`方法新增数据默认会自动识别数据是需要新增还是更新操作，当数据中存在主键的时候会认为是更新操作，如果你需要带主键数据批量新增，可以使用下面的方式：

```
$user = new User;
$list = [
    ['id'=>1, 'name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['id'=>2, 'name'=>'onethink', 'email'=>'onethink@qq.com'],
];
$user->saveAll($list, false);
```

静态方法

还可以直接静态调用`create`方法创建并写入：

```
$user = User::create([
    'name'  =>  'thinkphp',
    'email' =>  'thinkphp@qq.com'
]);
echo $user->name;
echo $user->email;
echo $user->id; // 获取自增ID
```

> 和`save`方法不同的是，`create`方法返回的是当前模型的对象实例。

`create`方法的第二个参数可以传入允许写入的字段列表（传入`true`则表示仅允许写入数据表定义的字段数据），例如：

```
// 只允许写入name和email字段的数据
$user = User::create([
    'name'  =>  'thinkphp',
    'email' =>  'thinkphp@qq.com'
], ['name', 'email']);
echo $user->name;
echo $user->email;
echo $user->id; // 获取自增ID
```

最佳实践

> 新增数据的最佳实践原则：使用 `create` 方法新增数据，使用 `saveAll` 批量新增数据。

更新

和模型新增一样，更新操作同样也会经过修改器、自动完成以及模型事件等处理，并不等同于数据库的数据更新，而且更新方法和新增方法使用的是同一个方法，通常系统会自动判断需要新增还是更新数据。

查找并更新

在取出数据后，更改字段内容后使用`save`方法更新数据。这种方式是最佳的更新方式。

```
$user = User::get(1);
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
```

> `save`方法返回影响的记录数，并只有当`before_update`事件返回`false`的时候返回`false`

对于复杂的查询条件，也可以使用查询构造器来查询数据并更新

```
$user = User::where('status',1)
        ->where('name','liuchen')
        ->find();
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
```

直接更新数据

也可以直接带更新条件来更新数据

```
$user = new User;
// save方法第二个参数为更新条件
$user->save([
    'name'  => 'thinkphp',
    'email' => 'thinkphp@qq.com'
],['id' => 1]);
```

上面两种方式更新数据，如果需要过滤非数据表字段的数据，可以使用：

```
$user = new User;
// 过滤post数组中的非数据表字段数据
$user->allowField(true)->save($_POST,['id' => 1]);
```

如果你通过外部提交赋值给模型，并且希望指定某些字段写入，可以使用：

```
$user = new User();
// post数组中只有name和email字段会写入
$user->allowField(['name','email'])->save($_POST, ['id' => 1]);
```

最佳建议是在传入模型数据之前就进行过滤，例如：

```
$user = new User();
// post数组中只有name和email字段会写入
$data = Request::only(['name','email']);
$user->save($data, ['id' => 1]);
```

批量更新数据

可以使用 `saveAll` 方法批量更新数据，只需要在批量更新的数据中包含主键即可，例如：

```
$user = new User;
$list = [
    ['id'=>1, 'name'=>'thinkphp', 'email'=>'thinkphp@qq.com'],
    ['id'=>2, 'name'=>'onethink', 'email'=>'onethink@qq.com']
];
$user->saveAll($list);
```

批量更新方法返回的是一个数据集对象。

> 批量更新仅能根据主键值进行更新，其它情况请自行处理。

静态方法

模型支持调用数据库的方法直接更新数据，例如：

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

> 数据库的 `update` 方法返回影响的记录数

或者使用模型的静态`update`方法更新：

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

> 模型的`update`方法返回模型的对象实例

> 上面两种写法的区别是第一种是使用的数据库的 update 方法，而第二种是使用的模型的 update 方法（可以支持模型的修改器、事件和自动完成）。

自动识别

我们已经看到，模型的新增和更新方法都是`save`方法，系统有一套默认的规则来识别当前的数据需要更新还是新增。

- 实例化模型后调用`save`方法表示新增；
- 查询数据后调用`save`方法表示更新；
- `save`方法传入更新条件后表示更新；

如果你的数据操作比较复杂，可以用`isUpdate`方法显式的指定当前调用`save`方法是新增操作还是更新操作。

显式更新数据：

```
// 实例化模型
$user = new User;
// 显式指定更新数据操作
$user->isUpdate(true)
    ->save(['id' => 1, 'name' => 'thinkphp']);
```

显式新增数据：

```
$user = User::get(1);
$user->name = 'thinkphp';
// 显式指定当前操作为新增操作
$user->isUpdate(false)->save();
```

不要在一个模型实例里面做多次更新，会导致部分重复数据不再更新，正确的方式应该是先查询后更新或者使用模型类的`update`方法更新。

> 如果你调用 save 方法进行多次数据写入的时候，需要注意，第二次 save 方法的时候必须使用 isUpdate(false) ，否则会视为更新数据。

最佳实践

> 更新的最佳实践原则是：如果需要使用模型事件，那么就先查询后更新，如果不需要使用事件，直接使用静态的 Update 方法进行条件更新，如非必要，尽量不要使用批量更新。

删除

模型的删除和数据库的删除方法区别在于，模型的删除会包含模型的事件处理。

删除当前模型

删除模型数据，可以在查询后调用 `delete` 方法。

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

> `delete` 方法返回影响的记录数

根据主键删除

或者直接调用静态方法（根据主键删除）

```
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]);
```

> 当 `destroy` 方法传入空值（包括空字符串和空数组）的时候不会做任何的数据删除操作，但传入0则是有效的

条件删除

还支持使用闭包删除，例如：

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

或者通过数据库类的查询条件删除

```
User::where('id','>',10)->delete();
```

> 直接调用数据库的 `delete` 方法的话无法调用模型事件。

最佳实践

> 删除的最佳实践原则是：如果删除当前模型数据，用 `delete` 方法，如果需要直接删除数据，使用 `destroy` 静态方法。

查询

模型查询和数据库查询方法的区别主要在于，模型中的查询的数据在获取的时候会经过获取器的处理，以及更加对象化的获取方式。

> 模型查询除了使用自身的查询方法外，一样可以使用数据库的查询构造器，返回的都是模型对象实例。

获取单个数据

获取单个数据的方法包括：

```
// 取出主键为1的数据
$user = User::get(1);
echo $user->name;

// 使用查询构造器查询满足条件的数据
$user = User::where('name', 'thinkphp')->find();
echo $user->name;
```

模型无论使用`get`还是`find`方法查询，返回的是都当前模型的对象实例，除了获取模型数据外，还可以使用模型的方法。

> `V5.1.5+`版本，模型增加了`getOrFail`方法用于查询，当查询的数据不存在的时候会抛出`ModelNotFound`异常。

> 如果你是在模型内部获取模型数据，请不要使用`$this->name`的方式来获取数据，请使用`$this->getAttr('name')`替代。

获取多个数据

取出多个数据：

```
// 根据主键获取多个数据
$list = User::all('1,2,3');
// 或者使用数组
$list = User::all([1,2,3]);
// 对数据集进行遍历操作
foreach($list as $key=>$user){
```

```
    echo $user->name;
}
```

要更多的查询支持，一样可以使用查询构造器：

```
// 使用查询构造器查询
$list = User::where('status', 1)->limit(3)->order('id', 'asc')->select();
foreach($list as $key=>$user){
    echo $user->name;
}
```

>查询构造器方式的查询可以支持更多的连贯操作，包括排序、数量限制等。

自定义数据集对象

模型的`all`方法或者`select`方法返回的是一个包含多个模型实例的数据集对象（默认为`\think\model\Collection`），支持在模型中单独设置查询数据集的返回对象的名称，例如：

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

use think\Model;

class User extends Model
{
        // 设置返回数据集的对象名
        protected $resultSetType = '\app\common\Collection';
}
```

`resultSetType`属性用于设置自定义的数据集使用的类名，该类应当继承系统的`think\model\Collection`类。

使用查询构造器

在模型中仍然可以调用数据库的链式操作和查询方法，可以充分利用数据库的查询构造器的优势。

例如：

```
User::where('id',10)->find();
User::where('status',1)->order('id desc')->select();
User::where('status',1)->limit(10)->select();
```

使用查询构造器直接使用静态方法调用即可，无需先实例化模型。

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');
```

> value 和 column 方法返回的不再是一个模型对象实例，而是纯粹的值或者某个列的数组。

动态查询

支持数据库的动态查询方法，例如：

```
// 根据name字段查询用户
$user = User::getByName('thinkphp');

// 根据email字段查询用户
$user = User::getByEmail('thinkphp@qq.com');
```

聚合查询

同样在模型中也可以调用数据库的聚合方法查询，例如：

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

数据分批处理

模型也可以支持对返回的数据分批处理，这在处理大量数据的时候非常有用，例如：

```
User::chunk(100,function($users) {
    foreach($users as $user){
        // 处理user模型对象
    }
});
```

使用游标查询

模型也可以使用数据库的`cursor`方法进行游标查询，返回生成器对象

```
foreach(User::where('status', 1)->cursor() as $user){
        echo $user->name;
}
```

`user`变量是一个模型对象实例。

查询缓存

`get`方法和`all`方法的支持使用查询缓存，可以直接在第二个参数传入true表示开启查询缓存。

```
$user = User::get(1,true);
$list  = User::all('1,2,3',true);
```

如果要设置缓存标识，则必须在第三个参数传入缓存标识。

```
$user = User::get(1,'','user');
$list  = User::all('1,2,3','','user_list');
```

最佳实践

> 模型查询的最佳实践原则是：在模型外部使用静态方法进行查询，内部使用动态方法查询，包括使用数据库的查询构造器。模型的查询始终返回对象实例，但可以和数组一样使用。

JSON数据字段

本章内容为 `V5.1.4+` 版本开始支持，可以更为方便的操作模型的JSON数据字段。

> 这里指的JSON数据包括JSON类型以及JSON格式（但并不是JSON类型字段）的数据

我们修改下User模型类

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

use think\Model;
class User extends Model
{
        // 设置json类型字段
        protected $json = ['info'];
}
```

定义后，可以进行如下JSON数据操作。

写入JSON数据

使用数组方式写入JSON数据：

```
$user = new User;
$user->name = 'thinkphp';
$user->info = [
        'email'    => 'thinkphp@qq.com',
    'nickname '=> '流年',
];
$user->save();
```

使用对象方式写入JSON数据

```
$user = new User;
$user->name = 'thinkphp';
$info = new StdClass();
$info->email = 'thinkphp@qq.com';
$info->nickname = '流年';
$user->info = $info;
$user->save();
```

查询JSON数据

```
$user = User::get(1);
echo $user->name; // thinkphp
echo $user->info->email; // thinkphp@qq.com
echo $user->info->nickname; // 流年
```

查询条件为JSON数据

```
$user = User::where('info->nickname','流年')->find();
echo $user->name; // thinkphp
echo $user->info->email; // thinkphp@qq.com
echo $user->info->nickname; // 流年
```

更新JSON数据

```
$user = User::get(1);
$user->name = 'kancloud';
$user->info->email = 'kancloud@qq.com';
$user->info->nickname = 'kancloud';
$user->save();
```

获取器

获取器

获取器的作用是对模型实例的（原始）数据做出自动处理。一个获取器对应模型的一个特殊方法，方法格式为：

> getFieldNameAttr

`FieldName` 为数据表字段的驼峰转换，定义了获取器之后会在下列情况自动触发：

- 模型的数据对象取值操作（`$model->field_name`）；
- 模型的序列化输出操作（`$model->toArray()` 及 `toJson()`）；
- 显式调用 `getAttr` 方法（`$this->getAttr('field_name')`）；

获取器的场景包括：

- 时间日期字段的格式化输出；
- 集合或枚举类型的输出；
- 数字状态字段的输出；
- 组合字段的输出；

例如，我们需要对状态值进行转换，可以使用：

```
<?php
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：

```
<?php
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']];
    }
}
```

> 获取器方法的第二个参数传入的是当前的所有数据数组。

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”
```

获取原始数据

如果你定义了获取器的情况下，希望获取数据表中的原始数据，可以使用：

```
$user = User::get(1);
// 通过获取器获取字段
echo $user->status;
// 获取原始字段数据
echo $user->getData('status');
// 获取全部原始数据
dump($user->getData());
```

修改器

修改器

和获取器相反，修改器的主要作用是对模型设置的数据对象值进行处理。

修改器方法的命名规范为：

> setFieldNameAttr

修改器的使用场景和读取器类似：

- 时间日期字段的转换写入；
- 集合或枚举类型的写入；
- 数字状态字段的写入；
- 某个字段涉及其它字段的条件或者组合写入；

定义了修改器之后会在下列情况下触发：

- 模型对象赋值；
- 调用模型的 `data` 方法，并且第二个参数传入 `true` ；
- 调用模型的 `save` 方法，并且传入数据；
- 显式调用模型的 `setAttr` 方法；
- 定义了该字段的自动完成；

例如：

```
<?php
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 setSerializeAttr($value,$data)
    {
        return serialize($data);
    }
}
```

> 修改器方法的第二个参数会自动传入当前的所有数据数组。

批量修改

除了赋值的方式可以触发修改器外，还可以用下面的方法批量触发修改器：

```
$user = new User();
$data['name'] = 'THINKPHP';
$data['email'] = 'thinkphp@qq.com';
$user->data($data, true);
$user->save();
echo $user->name; // thinkphp
```

如果为 `name` 和 `email` 字段都定义了修改器的话，都会进行处理。

或者直接使用save方法触发，例如：

```
$user = new User();
$data['name'] = 'THINKPHP';
$data['email'] = 'thinkphp@qq.com';
$user->save($data);
echo $user->name; // thinkphp
```

> 修改器方法仅对模型的写入方法有效，调用数据库的写入方法写入无效，例如下面的方式修改器无效。
>
> ```
> $user = new User();
> $data['name'] = 'THINKPHP';
> $data['email'] = 'thinkphp@qq.com';
> $user->insert($data);
> ```

自动时间戳

系统支持自动写入创建和更新的时间戳字段（默认关闭），有两种方式配置支持。

第一种方式是全局开启，在数据库配置文件中进行设置：

```
// 开启自动写入时间戳字段
'auto_timestamp' => true,
```

第二种是在需要的模型类里面单独开启：

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

use think\Model;

class User extends Model
{
    protected $autoWriteTimestamp = true;
}
```

又或者首先在数据库配置文件中全局开启，然后在个别不需要使用自动时间戳写入的模型类中单独关闭：

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

use think\Model;

class User extends Model
{
    protected $autoWriteTimestamp = false;
}
```

一旦配置开启的话，会自动写入`create_time`和`update_time`两个字段的值，默认为整型（`int`），如果你的时间字段不是`int`类型的话，可以直接使用：

```
// 开启自动写入时间戳字段
'auto_timestamp' => 'datetime',
```

或者

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

use think\Model;

class User extends Model
{
    protected $autoWriteTimestamp = 'datetime';
}
```

默认的创建时间字段为`create_time`，更新时间字段为`update_time`，支持的字段类型包括`timestamp/datetime/int`。

写入数据的时候，系统会自动写入`create_time`和`update_time`字段，而不需要定义修改器，例如：

```
$user = new User();
$user->name = 'thinkphp';
$user->save();
echo $user->create_time; // 输出类似 2016-10-12 14:20:10
echo $user->update_time; // 输出类似 2016-10-12 14:20:10
```

> 时间字段的自动写入仅针对模型的写入方法，如果使用数据库的更新或者写入方法则无效。

> 时间字段输出的时候会自动进行格式转换，如果不希望自动格式化输出，可以把数据库配置文件的 `datetime_format` 参数值改为 `false`

`datetime_format`参数支持设置为一个时间类名，这样便于你进行更多的时间处理，例如：

```
// 设置时间字段的格式化类
'datetime_format' => '\org\util\DateTime',
```

该类应该包含一个`__toString`方法定义以确保能正常写入数据库。

如果你的数据表字段不是默认值的话，可以按照下面的方式定义：

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

```
use think\Model;

class User extends Model
{
    // 定义时间戳字段名
    protected $createTime = 'create_at';
    protected $updateTime = 'update_at';
}
```

下面是修改字段后的输出代码：

```
$user = new User();
$user->name = 'thinkphp';
$user->save();
echo $user->create_at; // 输出类似 2016-10-12 14:20:10
echo $user->update_at; // 输出类似 2016-10-12 14:20:10
```

如果你只需要使用 `create_time` 字段而不需要自动写入 `update_time` ，则可以单独关闭某个字段，例如：

```
class User extends Model
{
    // 关闭自动写入update_time字段
    protected $updateTime = false;
}
```

支持动态关闭时间戳写入功能，例如你希望更新阅读数的时候不修改更新时间，可以使用 `isAutoWriteTimestamp` 方法：

```
$user = User::get(1);
$user->read +=1;
$user->isAutoWriteTimestamp(false)->save();
```

只读字段

只读字段用来保护某些特殊的字段值不被更改，这个字段的值一旦写入，就无法更改。 要使用只读字段的功能，我们只需要在模型中定义`readonly`属性：

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

use think\Model;

class User extends Model
{
    protected $readonly = ['name', 'email'];
}
```

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

下面举个例子说明下：

```
$user = User::get(5);
 // 更改某些字段的值
$user->name = 'TOPThink';
$user->email = 'Topthink@gmail.com';
$user->address = '上海静安区';
 // 保存更改后的用户数据
$user->save();
```

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

5.1版本支持动态设置只读字段，例如：

```
$user = User::get(5);
 // 更改某些字段的值
$user->name = 'TOPThink';
$user->email = 'Topthink@gmail.com';
$user->address = '上海静安区';
 // 保存更改后的用户数据
$user->readonly(['name','email'])->save();
```

> 只读字段仅针对模型的更新方法，如果使用数据库的更新方法则无效，例如下面的方式无效。

```
$user = new User;
 // 要更改字段值
$data['name'] = 'TOPThink';
$data['email'] = 'Topthink@gmail.com';
$data['address'] = '上海静安区';
 // 保存更改后的用户数据
$user->where('id', 5)->update($data);
```

软删除

软删除

在实际项目中，对数据频繁使用删除操作会导致性能问题，软删除的作用就是把数据加上删除标记，而不是真正的删除，同时也便于需要的时候进行数据的恢复。

要使用软删除功能，需要引入`SoftDelete` trait，例如`User`模型按照下面的定义就可以使用软删除功能：

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

use think\Model;
use think\model\concern\SoftDelete;

class User extends Model
{
    use SoftDelete;
    protected $deleteTime = 'delete_time';
}
```

`deleteTime`属性用于定义你的软删除标记字段，`ThinkPHP`的软删除功能使用时间戳类型（数据表默认值为`Null`），用于记录数据的删除时间。

> 可以用类型转换指定软删除字段的类型，建议数据表的所有时间字段统一一种类型。

定义好模型后，我们就可以使用：

```
// 软删除
User::destroy(1);
// 真实删除
User::destroy(1,true);

$user = User::get(1);
// 软删除
$user->delete();
// 真实删除
$user->delete(true);
```

默认情况下查询的数据不包含软删除数据，如果需要包含软删除的数据，可以使用下面的方式查询：

```
User::withTrashed()->find();
User::withTrashed()->select();
```

如果仅仅需要查询软删除的数据，可以使用：

```
User::onlyTrashed()->find();
User::onlyTrashed()->select();
```

恢复被软删除的数据

```
$user = User::onlyTrashed()->find(1);
$user->restore();
```

> 软删除仅对模型的删除方法有效，如果直接使用数据库的删除方法则无效，例如下面的方式无效（将不会执行任何操作）。

```
$user = new User;
$user->where('id',1)->delete();
```

类型转换

支持给字段设置类型自动转换，会在写入和读取的时候自动进行类型转换处理，例如：

```
<?php
class User extends Model
{
    protected $type = [
        'status'    =>  'integer',
        'score'     =>  'float',
        'birthday'  =>  'datetime',
        'info'      =>  'array',
    ];
}
```

下面是一个类型自动转换的示例：

```
$user = new User;
$user->status = '1';
$user->score = '90.50';
$user->birthday = '2015/5/1';
$user->info = ['a'=>1,'b'=>2];
$user->save();
var_dump($user->status); // int 1
var_dump($user->score); // float 90.5;
var_dump($user->birthday); // string '2015-05-01 00:00:00'
var_dump($user->info);// array (size=2) 'a' => int 1  'b' => int 2
```

数据库查询默认取出来的数据都是字符串类型，如果需要转换为其他的类型，需要设置，支持的类型包括如下类型：

`integer`

设置为integer（整型）后，该字段写入和输出的时候都会自动转换为整型。

`float`

该字段的值写入和输出的时候自动转换为浮点型。

`boolean`

该字段的值写入和输出的时候自动转换为布尔型。

`array`

如果设置为强制转换为`array`类型，系统会自动把数组编码为json格式字符串写入数据库，取出来的时候会自动解码。

object

该字段的值在写入的时候会自动编码为json字符串，输出的时候会自动转换为`stdclass`对象。

serialize

指定为序列化类型的话，数据会自动序列化写入，并且在读取的时候自动反序列化。

json

指定为`json`类型的话，数据会自动`json_encode`写入，并且在读取的时候自动`json_decode`处理。

timestamp

指定为时间戳字段类型的话，该字段的值在写入时候会自动使用`strtotime`生成对应的时间戳，输出的时候会自动转换为`dateFormat`属性定义的时间字符串格式，默认的格式为`Y-m-d H:i:s`，如果希望改变其他格式，可以定义如下：

```
<?php
class User extends Model
{
    protected $dateFormat = 'Y/m/d';
    protected $type = [
        'status'    =>  'integer',
        'score'     =>  'float',
        'birthday'  =>  'timestamp',
    ];
}
```

或者在类型转换定义的时候使用：

```
<?php
class User extends Model
{
    protected $type = [
        'status'    =>  'integer',
        'score'     =>  'float',
        'birthday'  =>  'timestamp:Y/m/d',
    ];
}
```

然后就可以

```
$user = User::find(1);
echo $user->birthday; // 2015/5/1
```

`datetime`

和`timestamp`类似，区别在于写入和读取数据的时候都会自动处理成时间字符串`Y-m-d H:i:s`的格式。

数据完成

数据自动完成指在不需要手动赋值的情况下对字段的值进行处理后写入数据库。

系统支持`auto`、`insert`和`update`三个属性，可以分别在写入、新增和更新的时候进行字段的自动完成机制，`auto`属性自动完成包含新增和更新操作，例如我们定义`User`模型类如下：

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

use think\Model;

class User extends Model
{
    protected $auto = ['name', 'ip'];
    protected $insert = ['status' => 1];
    protected $update = [];

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

    protected function setIpAttr()
    {
        return request()->ip();
    }
}
```

> 数据自动完成如果需要写入固定的值，可以直接指定（例如上面的status字段固定写入了1），类似于数据表字段的默认值功能。

在新增数据的时候，会对`name`、`ip`和`status`字段自动完成或者处理。

```
$user = new User;
$user->name = 'ThinkPHP';
$user->save();
echo $user->name; // thinkphp
echo $user->status; // 1
```

在更新数据的时候，会自动处理`name`字段的值及完成`ip`字段的赋值。

```
$user = User::find(1);
$user->name = 'THINKPHP';
$user->save();
echo $user->name; // thinkphp
echo $user->ip; // 127.0.0.1
```

> 数据自动完成仍然还是调用的修改器，要注意避免数据被两次处理的可能，自动完成定义的属性不要和表单提交的冲突。

查询范围

可以对模型的查询和写入操作进行封装，例如：

```
<?php
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')->find();
// 查找年龄大于20的10个用户
User::scope('age')->select();
// 查找name为thinkphp的用户并且年龄大于20的10个用户
User::scope('thinkphp,age')->select();
```

查询范围的方法可以定义额外的参数，例如User模型类定义如下：

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

use think\Model;

class User extends Model
{
        public function scopeEmail($query, $email)
    {
        $query->where('email', 'like', '%' . $email . '%');
    }
```

```
    public function scopeScore($query, $score)
    {
        $query->where('score', '>', $score);
    }

}
```

在查询的时候可以如下使用：

```
// 查询email包含thinkphp和分数大于80的用户
User::email('thinkphp')->score(80)->select();
```

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);
})->select();
```

> 使用查询范围后，只能使用 find 或者 select 查询。

全局查询范围

如果你的所有查询都需要一个基础的查询范围，那么可以在模型类里面定义一个静态的 `base` 方法，例如：

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

use think\Model;

class User extends Model
{
    // 定义全局的查询范围
    protected function base($query)
    {
        $query->where('status',1);
    }
}
```

然后，执行下面的代码：

```
$user = User::get(1);
```

最终的查询条件会是

```
status = 1 AND id = 1
```

如果需要动态关闭/开启全局查询访问，可以使用：

```
// 关闭全局查询范围
User::useGlobalScope(false)->get(1);
// 开启全局查询范围
User::useGlobalScope(true)->get(2);
```

模型输出

模板输出

模型数据的模板输出可以直接把模型对象实例赋值给模板变量，在模板中可以直接输出，例如：

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

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

class Index extends Controller
{
    public function index()
    {
        $user = User::find(1);
        $this->assign('user', $user);

        return $this->fetch();
    }
}
```

在模板文件中可以使用

```
{$user.name}
{$user.email}
```

> 模板中的模型数据输出一样会调用获取器。

数组转换

可以使用 `toArray` 方法将当前的模型实例输出为数组，例如：

```
$user = User::find(1);
dump($user->toArray());
```

支持设置不输出的字段属性：

```
$user = User::find(1);
dump($user->hidden(['create_time','update_time'])->toArray());
```

数组输出的字段值会经过获取器的处理，也可以支持追加其它获取器定义（不在数据表字段列表中）的字段，例如：

```
$user = User::find(1);
dump($user->append(['status_text'])->toArray());
```

支持设置允许输出的属性，例如：

```
$user = User::find(1);
dump($user->visible(['id','name','email'])->toArray());
```

对于数据集结果一样可以直接使用（包括`append`、`visible`和`hidden`方法）

```
$list = User::all();
$list = $list->toArray();
```

追加关联属性

支持追加一对一关联模型的属性到当前模型，例如：

```
$user = User::find(1);
dump($user->append(['profile' => ['email', 'nickname']])->toArray());
```

`profile`是关联定义方法名，`email`和`nickname`是`Profile`模型的属性。

模型的`visible`、`hidden`和`append`方法支持关联属性操作，例如：

```
$user = User::get(1,'profile');
// 隐藏profile关联属性的email属性
dump($user->hidden(['profile'=>['email']])->toArray());
// 或者使用
dump($user->hidden(['profile.email'])->toArray());
```

`hidden`、`visible`和`append`方法同样支持数据集对象。

JSON序列化

可以调用模型的`toJson`方法进行`JSON`序列化，`toJson`方法的使用和`toArray`一样。

```
$user = User::get(1);
echo $user->toJson();
```

可以设置需要隐藏的字段，例如：

```
$user = User::get(1);
echo $user->hidden(['create_time','update_time'])->toJson();
```

或者追加其它的字段（该字段必须有定义获取器）：

```
$user = User::get(1);
echo $user->append(['status_text'])->toJson();
```

设置允许输出的属性：

```
$user = User::get(1);
echo $user->visible(['id','name','email'])->toJson();
```

模型对象可以直接被JSON序列化，例如：

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

输出结果类似于：

```
{"id":"1","name":"","title":"","status":"1","update_time":"1430409600","s
core":"90.5"}
```

如果直接`echo`一个模型对象会自动调用模型的`toJson`方法输出，例如：

```
echo User::get(1);
```

输出的结果和上面是一样的。

事件

模型事件

模型事件是指在进行模型的写入操作的时候触发的操作行为，包括模型的`save`方法和`delete`方法。

> 模型事件只在调用模型的方法生效，使用查询构造器操作是无效的

模型支持如下事件：

事件	描述	快捷方法
before_insert	新增前	beforeInsert
after_insert	新增后	afterInsert
before_update	更新前	beforeUpdate
after_update	更新后	afterUpdate
before_write	写入前	beforeWrite
after_write	写入后	afterWrite
before_delete	删除前	beforeDelete
after_delete	删除后	afterDelete

使用方法如下：

```
<?php

namespace app\index\model;

use think\Model;

class User extends Model
{
    public static function init()
    {
        self::event('before_insert', function ($user) {
            if (1 != $user->status) {
                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），但支持依赖注入的方式增加额外参数。

并且`before_write`、`before_insert`、`before_update`、`before_delete`事件方法如果返回false，则不会继续执行。

支持给一个位置注册多个回调方法，例如：

```
        User::event('before_insert', function ($user) {
            if ($user->status != 1) {
                return false;
            }
        });
        // 注册回调到beforeInsert函数
        User::event('before_insert', 'beforeInsert');
```

快捷注册

系统提供了内置的事件注册的快捷方法，你可以统一在init方法中进行模型事件定义：

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

use think\Model;

class User extends Model
{
    protected static function init()
    {
        self::beforeInsert(function ($user) {
            if ($user->status != 1) {
                return false;
            }
        });
    }
}
```

关联

模型关联

通过模型关联操作把数据表的关联关系对象化，解决了大部分常用的关联场景，封装的关联操作比起常规的数据库联表操作更加智能和高效，并且直观。

> 避免在模型内部使用复杂的 join 查询和视图查询。

从面向对象的角度来看关联的话，模型的关联其实应该是模型的某个属性，比如用户的档案关联，就应该是下面的情况：

```
// 获取用户模型实例
$user = User::get(1);
// 获取用户的档案
$user->profile;
// 获取用户的档案中的手机资料
$user->profile->mobile;
```

为了更方便和灵活的定义模型的关联关系，框架选择了方法定义而不是属性定义的方式，每个关联属性其实是对应了一个模型的（关联）方法，这个关联属性和模型的数据一样是动态的，并非模型类的实体属性。

例如上面的关联属性就是在`User`模型类中定义了一个`profile`方法（`mobile`属性是`Profile`模型的属性）：

```
<?php

namespace app\index\model;

use think\Model;

class User extends Model
{
        public function profile()
    {
        return $this->hasOne('Profile');
    }
}
```

> 一个模型可以定义多个不同的关联，增加不同的关联方法即可

同时，我们必须定义一个 `Profile` 模型（即使是一个空模型）。

```
<?php

namespace app\index\model;

use think\Model;

class Profile extends Model
{
}
```

关联方法返回的是不同的关联对象，例如这里的 `profile` 方法返回的是一个 `HasOne` 关联对象（ `think\model\relation\HasOne` ）实例。

当我们访问 `User` 模型对象实例的 `profile` 属性的时候，其实就是调用了 `profile` 方法来完成关联查询。

按照 `PSR-2` 规范，模型的方法名都是驼峰命名的，所以系统做了一个兼容处理，如果我们定义了一个 `userProfile` 的关联方法的时候，在获取关联属性的时候，下面两种方式都是有效的：

```
$user->userProfile;
$user->user_profile;
```

> 推荐关联属性统一使用后者，和数据表的字段命名规范一致，因此在很多时候系统自动获取关联属性的时候采用的也是后者。

可以简单的理解为关联定义就是在模型类中添加一个方法（注意不要和模型的对象属性以及其它业务逻辑方法冲突），一般情况下无需任何参数，并在方法中指定一种关联关系，比如上面的 `hasOne` 关联关系， `5.1` 版本支持的关联关系包括下面8种，后面会给大家陆续介绍：

模型方法	关联类型
`hasOne`	一对一
`belongsTo`	一对一
`hasMany`	一对多
`hasManyThrough`	远程一对多

`belongsToMany`	多对多
`morphMany`	多态一对多
`morphOne`	多态一对一
`morphTo`	多态

关联方法的第一个参数就是要关联的模型名称，也就是说当前模型的关联模型必须也是已经定义好的一个模型。

一般不需要使用命名空间，会自动使用当前模型的命名空间，如果不同请使用完整命名空间定义，例如：

```
<?php

namespace app\index\model;

use think\Model;

class User extends Model
{
        public function profile()
    {
        // Profile模型和当前模型的命名空间不一致
        return $this->hasOne('app\model\Profile');
    }
}
```

两个模型之间因为参照模型的不同就会产生相对的但不一定相同的关联关系，并且相对的关联关系只有在需要调用的时候才需要定义，下面是每个关联类型的相对关联关系对照：

类型	关联关系	相对的关联关系
一对一	`hasOne`	`belongsTo`
一对多	`hasMany`	`belongsTo`
多对多	`belongsToMany`	`belongsToMany`
远程一对多	`hasManyThrough`	不支持
多态一对一	`morphOne`	`morphTo`
多态一对多	`morphMany`	`morphTo`

例如，`Profile` 模型中就可以定义一个相对的关联关系。

```
<?php

namespace app\index\model;
```

```
use think\Model;

class Profile extends Model
{
        public function user()
    {
        return $this->belongsTo('User');
    }
}
```

在进行关联查询的时候，也是类似，只是当前模型不同。

```
// 获取档案实例
$profile = Profile::get(1);
// 获取档案所属的用户名称
echo $profile->user->name;
```

如果你需要对关联模型进行更多的查询约束，可以在关联方法的定义方法后面追加额外的查询链式方法（但切忌不要滥用，并且不要使用实际的查询方法），例如：

```
<?php

namespace app\index\model;

use think\Model;

class User extends Model
{
        public function book()
    {
        return $this->hasMany('Book')->order('pub_time');
    }
}
```

> 5.1版本的模型关联支持调用模型的方法

> 具体不同的关联关系的详细使用，请继续参考后面的内容。

一对一关联

一对一关联

版本	新增功能
5.1.2	增加 `selfRelation` 方法定义当前关联为自关联

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：

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

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile');
    }
}
```

`hasOne` 方法的参数包括：

> hasOne('关联模型','外键','主键');

除了关联模型外，其它参数都是可选。

- 关联模型（必须）：关联的模型名或者类名
- 外键：默认的外键规则是当前模型名（不含命名空间，下同）+ `_id` ，例如 `user_id`
- 主键：当前模型主键，默认会自动获取也可以指定传入

一对一关联定义的时候还支持额外的方法，包括：

方法名	描述
setEagerlyType	定义关联查询方式，0 - `JOIN` 查询 1 - `IN` 查询（默认）
bind	绑定关联属性到父模型
joinType	JOIN方式查询的JOIN方式，默认为 `INNER`
selfRelation	定义当前关联为自关联

> 如果使用了JOIN方式的关联查询方式，你可以在额外的查询条件中使用关联方法名作为表的别名。

关联查询

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::get(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：

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

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile','uid');
    }
}
```

> 有一点需要注意的是，关联方法的命名规范是驼峰法，而关联属性则一般是小写+下划线的方式，系统在获取的时候会自动转换对应，读取`user_profile`关联属性则对应的关联方法应该是`userProfile`。

关联保存

```
$user = User::get(1);
// 如果还没有关联数据 则进行新增
$user->profile()->save(['email' => 'thinkphp']);
```

系统会自动把当前模型的主键传入`Profile`模型。

和新增一样使用`save`方法进行更新关联数据。

```
$user = User::get(1);
```

```
$user->profile->email = 'thinkphp';
$user->profile->save();
// 或者
$user->profile->save(['email' => 'thinkphp']);
```

定义相对关联

我们可以在`Profile`模型中定义一个相对的关联关系，例如：

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

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo('User');
    }
}
```

`belongsTo`的参数包括：

> belongsTo('关联模型','外键','关联主键');

除了关联模型外，其它参数都是可选。

- 关联模型（必须）：模型名或者模型类名
- 外键：当前模型外键，默认的外键名规则是关联模型名+`_id`
- 关联主键：关联模型主键，一般会自动获取也可以指定传入

默认的关联外键是`user_id`，如果不是，需要在第二个参数定义

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

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo('User','uid');
    }
}
```

我们就可以根据档案资料来获取用户模型的信息

```
$profile = Profile::get(1);
// 输出User关联模型的属性
echo $profile->user->account;
```

绑定属性到父模型

可以在定义关联的时候使用`bind`方法绑定属性到父模型，例如：

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

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile','uid')->bind('nickname,email');
    }
}
```

或者使用数组的方式指定绑定属性别名

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

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne('Profile','uid')->bind([
                        'email',
                'truename'      => 'nickname',
                'profile_id'  => 'id',
            ]);
    }
}
```

然后使用关联预载入查询的时候，可以使用

```
$user = User::get(1,'profile');
// 输出Profile关联模型的email属性
echo $user->email;
```

```
echo $user->profile_id;
```

绑定关联模型的属性支持读取器。

> 如果不是预载入查询，请使用模型的`appendRelationAttr`方法追加属性。

关联自动写入

我们可以使用`together`方法更方便的进行关联自动写入操作。

写入

```
$blog = new Blog;
$blog->name = 'thinkphp';
$blog->title = 'ThinkPHP5关联实例';
$content = new Content;
$content->data = '实例内容';
$blog->content = $content;
$blog->together('content')->save();
```

如果绑定了子模型的属性到当前模型，可以用数组指定子模型的属性

```
$blog = new Blog;
$blog->name = 'thinkphp';
$blog->title = 'ThinkPHP5关联实例';
$blog->content = '实例内容';
// title和content是子模型的属性
$blog->together(['content'=>['title','content']])->save();
```

更新

```
// 查询
$blog = Blog::get(1);
$blog->title = '更改标题';
$blog->content->data = '更新内容';
// 更新当前模型及关联模型
$blog->together('content')->save();
```

删除

```
// 查询
$blog = Blog::get(1,'content');
// 删除当前及关联模型
$blog->together('content')->delete();
```

> 如果不想这么麻烦每次调用`together`方法，也可以直接在模型类中定义`relationWrite`属性，但必须是数组方式。不过考虑到模型的独立操作的可能性，并不建议。

一对多关联

一对多关联

关联定义

一对多关联的情况也比较常见，使用 `hasMany` 方法定义，参数包括：

> hasMany('关联模型','外键','主键');

除了关联模型外，其它参数都是可选。

- 关联模型（必须）：模型名或者模型类名
- 外键：关联模型外键，默认的外键名规则是当前模型名+`_id`
- 主键：当前模型主键，一般会自动获取也可以指定传入

例如一篇文章可以有多个评论

```
<?php
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

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

use think\Model;

class Article extends Model
{
    public function comments()
    {
        return $this->hasMany('Comment','art_id');
    }
}
```

关联查询

我们可以通过下面的方式获取关联数据

```
$article = Article::get(1);
// 获取文章的所有评论
dump($article->comments);
// 也可以进行条件搜索
dump($article->comments()->where('status',1)->select());
```

根据关联条件查询

可以根据关联条件来查询当前模型对象数据，例如：

```
// 查询评论超过3个的文章
$list = Article::has('comments','>',3)->select();
// 查询评论状态正常的文章
$list = Article::hasWhere('comments',['status'=>1])->select();
```

关联新增

```
$article = Article::find(1);
// 增加一个关联数据
$article->comments()->save(['content'=>'test']);
// 批量增加关联数据
$article->comments()->saveAll([
    ['content'=>'thinkphp'],
    ['content'=>'onethink'],
]);
```

定义相对的关联

要在 Comment 模型定义相对应的关联，可使用 `belongsTo` 方法：

```
<?php
name app\index\model;

use think\Model;

class Comment extends Model
{
    public function article()
    {
        return $this->belongsTo('article');
    }
}
```

关联删除

在删除文章的同时删除下面的评论

```
$article = Article::get(1,'comments');
$article->together('comments')->delete();
```

远程一对多

远程一对多关联用于定义有跨表的一对多关系，例如：

- 每个城市有多个用户
- 每个用户有多个话题
- 城市和话题之间并无关联

关联定义

就可以直接通过远程一对多关联获取每个城市的多个话题，`City`模型定义如下：

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

use think\Model;

class City extends Model
{
    public function topics()
    {
        return $this->hasManyThrough('Topic','User');
    }
}
```

远程一对多关联，需要同时存在`Topic`和`User`模型。

`hasManyThrough`方法的参数如下：

> hasManyThrough('关联模型','中间模型','外键','中间表关联键','主键');

- 关联模型（必须）：模型名或者模型类名
- 中间模型（必须）：模型名或者模型类名
- 外键：默认的外键名规则是当前模型名+`_id`
- 中间表关联键：默认的中间表关联键名的规则是中间模型名+`_id`
- 主键：当前模型主键，一般会自动获取也可以指定传入

关联查询

我们可以通过下面的方式获取关联数据

```
$city = City::get(1);
```

```
// 获取同城的所有话题
dump($city->topics);
// 也可以进行条件搜索
dump($city->topics()->where('topic.status',1)->select());
```

> 条件搜索的时候，需要带上模型名作为前缀

多对多关联

多对多关联
关联定义
例如，我们的用户和角色就是一种多对多的关系，我们在`User`模型定义如下：

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

use think\Model;

class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany('Role');
    }
}
```

`belongsToMany`方法的参数如下：

> belongsToMany('关联模型','中间表','外键','关联键');

- 关联模型（必须）：模型名或者模型类名
- 中间表：默认规则是当前模型名+`_`+关联模型名（可以指定模型名）
- 外键：中间表的当前模型外键，默认的外键名规则是关联模型名+`_id`
- 关联键：中间表的当前模型关联键名，默认规则是当前模型名+`_id`

中间表名无需添加表前缀，并支持定义中间表模型，例如：

```
    public function roles()
    {
        return $this->belongsToMany('Role','\\app\\model\\Access');
    }
```

中间表模型类必须继承`think\model\Pivot`，例如：

```
<?php
namespace app\index\model\Access;

use think\model\Pivot;
```

```
class Access extends Pivot
{
    protected $autoWriteTimestamp = true;
}
```

中间表模型的基类`Pivot`默认关闭了时间戳自动写入，上面的中间表模型则开启了时间戳字段自动写入。

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
$roles = $user->roles;
foreach ($roles as $role) {
        // 输出用户的角色名
        echo $role->name;
    // 获取中间表模型
    dump($role->pivot);
}
```

关联新增

```
$user = User::get(1);
// 给用户增加管理员权限 会自动写入角色表和中间表数据
$user->roles()->save(['name'=>'管理员']);
// 批量授权
$user->roles()->saveAll([
    ['name'=>'管理员'],
    ['name'=>'操作员'],
]);
```

只新增中间表数据（角色已经提前创建完成），可以使用

```
$user = User::get(1);
// 仅增加管理员权限（假设管理员的角色ID是1）
$user->roles()->save(1);
// 或者
$role = Role::get(1);
$user->roles()->save($role);
// 批量增加关联数据
$user->roles()->saveAll([1,2,3]);
```

单独更新中间表数据，可以使用：

```
$user = User::get(1);
// 增加关联的中间表数据
$user->roles()->attach(1);
// 传入中间表的额外属性
$user->roles()->attach(1,['remark'=>'test']);
// 删除中间表数据
$user->roles()->detach([1,2,3]);
```

> attach 方法的返回值是一个 Pivot 对象实例，如果是附加多个关联数据，则返回 Pivot 对象实例的数组。

定义相对的关联

我们可以在 Role 模型中定义一个相对的关联关系，例如：

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

use think\Model;

class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany('User');
    }
}
```

多态关联

多态一对多关联

多态关联允许一个模型在单个关联定义方法中从属一个以上其它模型，例如用户可以评论书和文章，但评论表通常都是同一个数据表的设计。多态一对多关联关系，就是为了满足类似的使用场景而设计。

下面是关联表的数据表结构：

```
article
    id - integer
    title - string
    content - text

book
    id - integer
    title - string

comment
    id - integer
    content - text
    commentable_id - integer
    commentable_type - string
```

有两个需要注意的字段是 `comment` 表中的 `commentable_id` 和 `commentable_type` 我们称之为多态字段。其中， `commentable_id` 用于存放书或者文章的 id（主键），而 `commentable_type` 用于存放所属模型的类型。通常的设计是多态字段有一个公共的前缀（例如这里用的 `commentable` ），当然，也支持设置完全不同的字段名（例如使用 `data_id` 和 `type` ）。

多态关联定义

接着，让我们来查看创建这种关联所需的模型定义：

文章模型：

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

use think\Model;

class Article extends Model
{
```

```
    /**
     * 获取所有针对文章的评论。
     */
    public function comments()
    {
        return $this->morphMany('Comment', 'commentable');
    }
}
```

`morphMany` 方法的参数如下：

> morphMany('关联模型','多态字段','多态类型');

关联模型（必须）：关联的模型名称，可以使用模型名（如 `Comment` ）或者完整的命名空间模型名（如 `app\index\model\Comment` ）。

多态字段（可选）：支持两种方式定义 如果是字符串表示多态字段的前缀，多态字段使用 `多态前缀_type` 和 `多态前缀_id` ，如果是数组，表示使用['多态类型字段名','多态ID字段名']，默认为当前的关联方法名作为字段前缀。

多态类型（可选）：当前模型对应的多态类型，默认为当前模型名，可以使用模型名（如 `Article` ）或者完整的命名空间模型名（如 `app\index\model\Article` ）。

书籍模型：

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

use think\Model;

class Book extends Model
{
    /**
     * 获取所有针对书籍的评论。
     */
    public function comments()
    {
        return $this->morphMany('Comment', 'commentable');
    }
}
```

书籍模型的设置方法同文章模型一致，区别在于多态类型不同，但由于多态类型默认会取当前模型名，因此不需要单独设置。

下面是评论模型的关联定义：

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

use think\Model;

class Comment extends Model
{
    /**
     * 获取评论对应的多态模型。
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}
```

`morphTo`方法的参数如下：

> morphTo('多态字段',['多态类型别名']);

多态字段（可选）：支持两种方式定义 如果是字符串表示多态字段的前缀，多态字段使用`多态前缀_type`和`多态前缀_id`，如果是数组，表示使用['多态类型字段名','多态ID字段名']，默认为当前的关联方法名作为字段前缀
多态类型别名（可选）：数组方式定义

获取多态关联

一旦你的数据表及模型被定义，则可以通过模型来访问关联。例如，若要访问某篇文章的所有评论，则可以简单的使用`comments`动态属性：

```
$article = Article::get(1);

foreach ($article->comments as $comment) {
    dump($comment);
}
```

你也可以从多态模型的多态关联中，通过访问调用`morphTo`的方法名称来获取拥有者，也就是此例子中`Comment`模型的`commentable`方法。所以，我们可以使用动态属性来访问这个方法：

```
$comment = Comment::get(1);
$commentable = $comment->commentable;
```

`Comment` 模型的 `commentable` 关联会返回 `Article` 或 `Book` 模型的对象实例，这取决于评论所属模型的类型。

自定义多态关联的类型字段

默认情况下，ThinkPHP 会使用模型名作为多态表的类型区分，例如，`Comment` 属于 `Article` 或者 `Book`，`commentable_type` 的默认值可以分别是 `Article` 或者 `Book`。我们可以通过定义多态的时候传入参数来对数据库进行解耦。

```
    public function commentable()
    {
        return $this->morphTo('commentable',[
                'book'  =>      'app\index\model\Book',
            'post'      =>      'app\admin\model\Article',
        ]);
    }
```

多态一对一关联

多态一对一相比多态一对多关联的区别是动态的一对一关联，举个例子说有一个个人和团队表，而无论个人还是团队都有一个头像需要保存但都会对应同一个头像表

```
member
    id - integer
    name - string

team
    id - integer
    name - string

avatar
    id - integer
    avatar - string
    imageable_id - integer
    imageable_type - string
```

会员模型：

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

use think\Model;

class Member extends Model
{
    /**
     * 获取用户的头像
```

```
     */
    public function avatar()
    {
        return $this->morphOne('Avatar', 'imageable');
    }
}
```

团队模型：

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

use think\Model;

class Team extends Model
{
    /**
     * 获取团队的头像
     */
    public function avatar()
    {
        return $this->morphOne('Avatar', 'imageable');
    }
}
```

`morphOne`方法的参数如下：

> morphOne('关联模型','多态字段','多态类型');

关联模型（必须）：关联的模型名称，可以使用模型名（如`Member`）或者完整的命名空间模型名（如`app\index\model\Member`）。

多态字段（可选）：支持两种方式定义 如果是字符串表示多态字段的前缀，多态字段使用`多态前缀_type`和`多态前缀_id`，如果是数组，表示使用['多态类型字段名','多态ID字段名']，默认为当前的关联方法名作为字段前缀。

多态类型（可选）：当前模型对应的多态类型，默认为当前模型名，可以使用模型名（如`Member`）或者完整的命名空间模型名（如`app\index\model\Member`）。

下面是头像模型的关联定义：

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

use think\Model;
```

```
class Avatar extends Model
{
    /**
     * 获取头像对应的多态模型。
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}
```

理解了多态一对多关联后，多态一对一关联其实就很容易理解了，区别就是当前模型和动态关联的模型之间的关联属于一对一关系。

关联预载入

关联预载入

关联查询的预查询载入功能，主要解决了`N+1`次查询的问题，例如下面的查询如果有3个记录，会执行4次查询：

```
$list = User::all([1,2,3]);
foreach($list as $user){
    // 获取用户关联的profile模型数据
    dump($user->profile);
}
```

如果使用关联预查询功能，就可以变成2次查询（对于一对一关联来说，如果使用JOIN方式只有一次查询），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);
}
```

支持使用数组方式定义嵌套预载入，例如下面的预载入要同时获取用户的`Profile`关联模型的`Phone`、`Job`和`Img`子关联模型数据：

```
$list = User::with(['profile'=>['phone','job','img']])->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联
```

```
    dump($user->profile->phone);
    dump($user->profile->job);
    dump($user->profile->img);
}
```

可以在模型的 `get` 和 `all` 方法中使用预载入，在第二个参数中传入预载入信息即可。例如下面的用法和使用 `with` 方法加 `select` 方法是等效的：

```
$list = User::all([1,2,3],'profile,book');
```

如果要指定属性查询，可以使用：

```
$list = User::field('id,name')->with(['profile'=>function($query){
        $query->field('email,phone');
}])->select([1,2,3]);

foreach($list as $user){
    // 获取用户关联的profile模型数据
    dump($user->profile);
}
```

关联预载入名称是关联方法名，支持传入方法名的小写和下划线定义方式，例如如果关联方法名是 `userProfile` 和 `userBook` 的话：

```
$list = User::with('userProfile,userBook')->select([1,2,3]);
```

等效于：

```
$list = User::with('user_profile,user_book')->select([1,2,3]);
```

一对一关联预载入支持两种方式：`JOIN` 方式（一次查询）和 `IN` 方式（两次查询，默认方式），如果要使用 `JOIN` 方式关联预载入，在关联定义方法中添加

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

use think\Model;

class User extends Model
{
    public function profile()
    {
```

```
        // 设置预载入查询方式为JOIN方式
        return $this->hasOne('Profile')->setEagerlyType(0);
    }
}
```

使用JOIN方式查询的话 关联定义的时候不能使用 `field` 方法指定字段，只能在预载入查询的时候使用 `withField` 方法指定字段，例如：

```
$list = User::with(['profile' => function($query){
        $query->withField('truename,email');
}])->select([1,2,3]);
```

延迟预载入

有些情况下，需要根据查询出来的数据来决定是否需要使用关联预载入，当然关联查询本身就能解决这个问题，因为关联查询是惰性的，不过用预载入的理由也很明显，性能具有优势。

延迟预载入仅针对多个数据的查询，因为单个数据的查询用延迟预载入和关联惰性查询没有任何区别，所以不需要使用延迟预载入。

如果你的数据集查询返回的是数据集对象，可以使用调用数据集对象的 `load` 实现延迟预载入：

```
// 查询数据集
$list = User::all([1,2,3]);
// 延迟预载入
$list->load('cards');
foreach($list as $user){
    // 获取用户关联的card模型数据
    dump($user->cards);
}
```

关联统计

关联统计

有些时候，并不需要获取关联数据，而只是希望获取关联数据的统计，这个时候可以使用 `withCount` 方法进行指定关联的统计。

```
$list = User::withCount('cards')->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的card关联统计
    echo $user->cards_count;
}
```

关联统计功能会在模型的对象属性中自动添加一个以“关联方法名+`_count`”（支持自定义）为名称的动态属性来保存相关的关联统计数据。

> 关联统计仅针对一对多或者多对多的关联关系，并且暂不支持远程一对多

如果需要对关联统计进行条件过滤，可以使用闭包方式。

```
$list = User::withCount(['cards'=>function($query){
    $query->where('status',1);
}])->select([1,2,3]);

foreach($list as $user){
    // 获取用户关联的card关联统计
    echo $user->cards_count;
}
```

> 一对一关联关系使用关联统计是无效的，一般可以用 `exists` 查询来判断是否存在关联数据。

支持给关联统计指定统计属性名，例如：

```
$list = User::withCount(['cards'=>'card_count'])->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的card关联统计
    echo $user->card_count;
}
```

和`withCount`类似的方法，还包括：

关联统计方法	描述
`withSum`	关联SUM统计
`withMax`	关联Max统计
`withMin`	关联Min统计
`withAvg`	关联Avg统计

除了`withCount`之外的统计方法需要指定统计字段名，用法如下：

```
$list = User::withSum('cards','total')->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的card关联余额统计
    echo $user->cards_sum;
}
```

同样，也可以指定统计字段名

```
$list = User::withSum(['cards'=>'card_total'],'total')->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联的card关联余额统计
    echo $user->card_total;
}
```

关联输出

关联数据的输出也可以使用`hidden`、`visible`和`append`方法进行控制，下面举例说明。

隐藏关联属性

如果要隐藏关联模型的属性，可以使用

```
$list = User::with('profile')->select();
$list->hidden(['profile.email'])->toArray());
```

输出的结果中就不会包含`Profile`模型的`email`属性，如果需要隐藏多个属性可以使用

```
$list = User::with('profile')->select();
$list->hidden(['profile'=>['address','phone','email'])->toArray());
```

显示关联属性

同样，可以使用visible方法来显示关联属性：

```
$list = User::with('profile')->select();
$list->visible(['profile'=>['address','phone','email'])->toArray());
```

追加关联属性

追加一个`Profile`模型的额外属性（非实际数据，可能是定义了获取器方法）

```
$list = User::with('profile')->select();
$list->append(['profile.status'])->toArray());
```

也可以追加一个额外关联对象的属性

```
$list = User::with('profile')->select();
$list->append(['Book.name'])->toArray());
```

视图

视图功能由 `\think\View` 类配合视图驱动（也即模板引擎驱动）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。

视图并非必须，尤其当使用接口模式开发的时候，你也可以在控制器中使用第三方的视图组件。

视图渲染

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。

模板渲染

渲染模板最常用的是控制器类在继承系统控制器基类（`\think\Controller`）后调用`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：

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

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 不带任何参数 自动定位当前操作的模板文件
        return $this->fetch();
    }
}
```

表示系统会按照默认规则自动定位模板文件，其规则是：

```
当前模块/view/当前控制器名（小写）/当前操作（小写）.html
```

如果有更改模板引擎的`view_depr`设置（假设`'view_depr'=>'_'`）的话，则上面的自动定位规则变成：

```
当前模块/view/当前控制器（小写）_当前操作（小写）.html
```

如果没有按照模板定义规则来定义模板文件（或者需要调用其他控制器下面的某个模板），可以使用：

```
// 指定模板输出
return $this->fetch('edit');
```

表示调用当前控制器下面的edit模板

```
return $this->fetch('member/read');
```

表示调用Member控制器下面的read模板。

跨模块渲染模板

```
return $this->fetch('admin@member/edit');
```

渲染输出不需要写模板文件的路径和后缀。这里面的控制器和操作并不一定需要有实际对应的控制器和操作，只是一个目录名称和文件名称而已，例如，你的项目里面可能根本没有Public控制器，更没有Public控制器的menu操作，但是一样可以使用

```
return $this->fetch('public/menu');
```

输出这个模板文件。理解了这个，模板输出就清晰了。

支持从视图根目录开始读取模板，例如：

```
return $this->fetch('/menu');
```

表示读取的模板是

```
当前模块/view/menu.html
```

> 如果需要调用视图类（ think\View)的其它方法，可以直接使用 $this->view 对

> 象。

如果你的模板文件位置比较特殊或者需要自定义模板文件的位置，可以采用下面的方式处理。

```
return $this->fetch('../template/public/menu.html');
```

这种方式需要带模板路径和后缀指定一个完整的模板文件位置，这里的`../template/public`目录是相对于当前项目入口文件位置。如果是其他的后缀文件，也支持直接输出，例如：

```
return $this->fetch('../template/public/menu.tpl');
```

只要`../template/public/menu.tpl`是一个实际存在的模板文件。

> 要注意模板文件位置是相对于应用的入口文件，而不是模板目录。

助手函数

如果你的控制器并未继承系统的控制器基类，则使用系统提供的助手函数`view`，可以完成相同的功能：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 渲染模板输出
        return view('hello', ['name' => 'thinkphp']);
    }
}
```

助手函数调用格式：

> view('[模板文件]'[,'模板变量（数组）'])

无论你是否继承`think\Controller`类，助手函数都可以使用，也是最方便的一种。

渲染内容

如果希望直接解析内容而不通过模板文件的话，可以使用 `display` 方法：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 直接渲染内容
        $content = '{$name}-{$email}';
        return $this->display($content, ['name' => 'thinkphp', 'email' =>
 'thinkphp@qq.com']);
    }
}
```

渲染的内容中一样可以使用模板引擎的相关标签。

视图赋值

模板赋值

除了系统变量和配置参数输出无需赋值外，其他变量如果需要在模板中输出必须首先进行模板赋值操作，否则会抛出异常，传递数据到模板输出有下面几种方式：

`assign`方法

在控制器继承了系统的控制器基类的情况下，可以使用`assign`方法进行模板变量赋值。

```
namespace index\app\controller;

class Index extends \think\Controller
{
    public function index()
    {
        // 模板变量赋值
        $this->assign('name','ThinkPHP');
        $this->assign('email','thinkphp@qq.com');
        // 或者批量赋值
        $this->assign([
            'name'  => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
        // 模板输出
        return $this->fetch('index');
    }
}
```

方法传入参数

方法`fetch`及`display`均可传入模板变量，例如

```
namespace app\index\controller;

class Index extends \think\Controller
{
    public function index()
    {
        return $this->fetch('index', [
            'name'  => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
    }
}
```

```
class Index extends \think\Controller
{
    public function index()
    {
        $content = '{$name}-{$email}';
        return $this->display($content, [
            'name'  => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
    }
}
```

助手函数

如果使用 `view` 助手函数渲染输出的话，可以使用下面的方法进行模板变量赋值：

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

或者

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

公共模板变量赋值

如果需要在控制器之外进行模板变量赋值，可以使用视图类的 `share` 静态方法进行全局公共模板变量赋值，例如：

```
use think\facade\View;
// 赋值全局模板变量
View::share('name','value');
// 或者批量赋值
View::share(['name1'=>'value','name2'=>'value2']);
```

全局静态模板变量最终会和前面使用方法赋值的模板变量合并。

视图过滤

视图过滤

可以对视图的渲染输出进行过滤

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

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 使用视图输出过滤
        return $this->filter(function($content){
                return str_replace("\r\n",'<br/>',$content);
        })->fetch();
    }
}
```

如果需要进行全局过滤，你可以在初始化方法中使用：

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

use think\Controller;

class Index extends Controller
{
        protected function initialize()
    {
        $this->view->filter(function($content){
                return str_replace("\r\n",'<br/>',$content);
        });
    }

    public function index()
    {
        // 使用视图输出过滤
        return $this->fetch();
    }
}
```

如果使用 `view` 助手函数进行模板渲染输出的话，可以使用下面的方式

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

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 使用视图输出过滤
        return view()->filter(function($content){
                return str_replace("\r\n",'<br/>',$content);
        });
    }
}
```

模板引擎

内置模板引擎

视图的模板文件可以支持不同的解析规则，默认情况下无需手动初始化模板引擎。

可以通过下面的几种方式对模板引擎进行初始化。

配置文件

内置模板引擎的参数统一在配置目录的`template.php`文件中配置，例如：

```
return [
    // 模板引擎类型 支持 php think 支持扩展
    'type'          => 'Think',
    // 模板路径
    'view_path'     => './template/',
    // 模板后缀
    'view_suffix'   => 'html',
    // 模板文件名分隔符
    'view_depr'     => DS,
    // 模板引擎普通标签开始标记
    'tpl_begin'     => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'       => '}',
    // 标签库标签开始标记
    'taglib_begin' => '{',
    // 标签库标签结束标记
    'taglib_end'    => '}',
],
```

视图根目录

视图文件的根目录默认情况下位于模块的`view`目录，每个模块的视图目录可以通过模板参数`view_path`自定义。

可以用`view_base`模板引擎参数定义全局的视图根目录，然后模块作为子目录。

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：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
```

```
    public function index()
    {
        // 使用内置PHP模板引擎渲染模板输出
        return $this->engine('php')->fetch();
    }
}
```

表示当前视图的模板文件使用原生php进行解析。

使用第三方模板引擎

官方扩展库中提供了一个类似于`angularjs`语法的模板引擎`think-angular`，具体可以参考参考手册。

另外还包括了一个`twig`模板引擎扩展：https://github.com/yunwuxin/think-twig

模板

本章的内容主要讲述了如何使用内置的模板引擎。

> 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。

> 由于编译型模板引擎的特性，模板缓存不能关闭，就算关闭缓存也会在每次渲染的时候重新生成模板缓存。

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、函数过滤和做一些基本的运算操作

XML标签	也称为标签库标签，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。

模板文件可以同时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 `template.php` 中的配置参数：

```
    // 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。

> 本手册后面的内容均使用默认的标签定界符配置进行说明

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。

5.1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：

```
    //标签库标签开始标签
    'taglib_begin'  =>  '<',
    //标签库标签结束标记
    'taglib_end'    =>  '>',
```

原来的

```
 {eq name="name" value="value"}
```

```
相等
{else/}
不相等
{/eq}
```

就需要改成

```
<eq name="name" value="value">
相等
<else/>
不相等
</eq>
```

变量输出

在模板中输出变量的方法很简单，例如，在控制器的方法中我们给模板变量赋值：

```
$this->assign('name', 'thinkphp');
return $this->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 htmlentities($name);?>!
```

这样，运行的时候就会在模板中显示：`Hello,ThinkPHP!`

注意模板标签的`{`和`$`之间不能有任何的空格，否则标签无效。所以，下面的标签

```
Hello,{ $name}!
```

将不会正常输出name变量，而是直接保持不变输出：`Hello,{ $name}!`

模板标签的变量输出根据变量类型有所区别，刚才我们输出的是字符串变量，如果是数组变量，

```
$data['name'] = 'ThinkPHP';
$data['email'] = 'thinkphp@qq.com';
$this->assign('data',$data);
```

那么，在模板中我们可以用下面的方式输出：

```
Name：{$data.name}
Email：{$data.email}
```

或者用下面的方式也是有效：

```
Name：{$data['name']}
Email：{$data['email']}
```

> 当我们要输出多维数组的时候，往往要采用后面一种方式。

如果`data`变量是一个对象（并且包含有`name`和`email`两个属性），那么可以用下面的方式输出：

```
Name：{$data->name}
Email：{$data->email}
```

也可以直接调用对象的常量或者方法

```
常量：{$data::CONST_NAME}
方法：{$data->fun()}
```

> 如果要输出模型数据的话，因为模型支持 `ArrayAccess` ，所以使用数组或者对象方式都可以输出。

使用默认值

我们可以给变量输出提供默认值，例如：

```
{$user.nickname|default="这家伙很懒，什么也没留下"}
```

对系统变量依然可以支持默认值输出，例如：

```
{$Think.get.name|default="名称为空"}
```

默认值和函数可以同时使用，例如：

```
{$Think.get.name|getName|default="名称为空"}
```

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.` （大小写一致）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
{$Think.get.page} // 输出$_GET['page']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.PHP_VERSION}
```

或者直接使用

```
{$Think.PHP_VERSION}
```

配置输出

输出配置参数使用：

```
{$Think.config.default_module}
{$Think.config.default_controller}
```

语言变量

输出语言变量可以使用：

```
{$Think.lang.page_error}
{$Think.lang.var_error}
```

请求变量

模板支持直接输出 `Request` 请求对象的方法参数，用法如下：

> $Request.方法名.参数

例如：

```
{$Request.get.id}
{$Request.param.name}
```

以 `$Request.` 开头的变量输出会认为是系统请求对象的参数输出。

> 支持 Request 类的大部分方法，但只支持方法的第一个参数。

下面都是有效的输出：

```
// 调用Request对象的get方法 传入参数为id
{$Request.get.id}
// 调用Request对象的param方法 传入参数为name
{$Request.param.name}
// 调用Request对象的param方法 传入参数为user.nickname
{$Request.param.user.nickname}
// 调用Request对象的root方法
{$Request.root}
// 调用Request对象的root方法，并且传入参数true
{$Request.root.true}
// 调用Request对象的path方法
{$Request.path}
// 调用Request对象的module方法
{$Request.module}
// 调用Request对象的controller方法
{$Request.controller}
// 调用Request对象的action方法
{$Request.action}
// 调用Request对象的ext方法
{$Request.ext}
// 调用Request对象的host方法
{$Request.host}
// 调用Request对象的ip方法
{$Request.ip}
// 调用Request对象的header方法
{$Request.header.accept-encoding}
```

使用函数

需要对模板输出使用函数进行过滤或其它处理的时候，可以使用：

```
{$data.name|md5}
```

可以使用空格，例如下面的写法是一样的：

```
{$data.name | md5}
```

编译后的结果是：

```
<?php echo htmlentities(md5($data['name'])); ?>
```

其中`htmlentities`方法是系统默认添加的（无需手动指定。

> 为了避免出现XSS安全问题，默认的变量输出都会使用`htmlentities`方法进行转义输出。

你还可以设置默认的过滤方法，在配置文件`template.php`中设置

```
'default_filter' => 'htmlspecialchars'
```

就会默认使用`htmlspecialchars`方法过滤输出。

如果你不需要转义（例如你需要输出html表格等内容），可以使用：

```
{$data.name|raw}
```

编译后的结果是：

```
<?php echo $data['name']; ?>
```

系统内置了下面几个固定的过滤规则（不区分大小写）

过滤方法	描述
date	日期格式化（支持各种时间类型）
format	字符串格式化
upper	转换为大写
lower	转换为小写
first	输出数组的第一个元素
last	输出数组的最后一个元素
default	默认值
raw	不使用（默认）转义

例如

```
{$data.create_time|date='Y-m-d H:i'}
{$data.number|format='%02d'}
```

如果函数有多个参数需要调用，可以使用

```
{$data.name|substr=0,3}
```

表示输出

```
<?php echo htmlentities(substr($data['name'],0,3)); ?>
```

还可以支持多个函数过滤，多个函数之间用“|”分割即可，例如：

```
{$name|md5|upper|substr=0,3}
```

编译后的结果是：

```
<?php echo htmlentities(substr(strtoupper(md5($name)),0,3)); ?>
```

函数会按照从左到右的顺序依次调用（系统默认的过滤规则会在最后调用）。

> 变量输出使用的函数可以支持内置的PHP函数或者用户自定义函数，甚至是静态方法。

如果你觉得这样写起来比较麻烦，也可以直接这样写：

```
{:substr(strtoupper(md5($name)),0,3)}
```

> 使用该方法输出的值不会使用默认的过滤方法进行转义。

可以在模板中直接使用系统的助手函数进行输出

```
{:app('cache')->get('name')}
```

表示调用容器中的 `think\Cache` 对象实例输出 `name` 缓存标识内容。

`{:` 开头的变量输出表示调用函数或者类的方法及属性，如果你要带命名空间调用类的属性，例如：

```
{:think\\App::VERSION}
{:think\\facade\\Request::get('name')}
```

> 类的命名空间中的 `\` 需要改成 `\\` 才能正常调用。

运算符

我们可以对模板输出使用运算符，包括如下支持。

运算符	使用示例
`+`	`{$a+$b}`
`-`	`{$a-$b}`
`*`	`{$a*$b}`
`/`	`{$a/$b}`
`%`	`{$a%$b}`
`++`	`{$a++}` 或 `{++$a}`
`--`	`{$a--}` 或 `{--$a}`
综合运算	`{$a+$b*10+$c}`

在使用运算符的时候，不再支持前面提到的函数过滤用法，例如：

```
{$user.score+10} //正确的
{$user['score']+10} //正确的
{$user['score']*$user['level']} //正确的
{$user['score']|myFun*10} //错误的
{$user['score']+myFun($user['level'])} //正确的
```

三元运算

模板可以支持三元运算符，例如：

```
{$status? '正常' : '错误'}
{$info['status']? $info['msg'] : $info['error']}
{$info.status? $info.msg : $info.error }
```

还支持如下的写法：

```
{$name ?? '默认值'}
```

表示如果有设置 `$name` 则输出 `$name` ,否则输出 `默认值` 。

```
{$name?='默认值'}
```

表示$name为真时才输出默认值。

```
{$name ?: 'NO'}
```

表示如果$name为真则输出$name，否则输出NO。

```
{$a==$b ? 'yes' : 'no'}
```

前面的表达式为真输出yes,否则输出no， 条件可以是==、===、!=、!==、>=、<=

原样输出

可以使用 `literal` 标签来防止模板标签被解析，例如：

```
{literal}
    Hello,{$name}!
{/literal}
```

上面的 `{$name}` 标签被 `literal` 标签包含，因此并不会被模板引擎解析，而是保持原样输出。

`literal` 标签还可以用于页面的JS代码外层，确保JS代码中的某些用法和模板引擎不产生混淆。

总之，所有可能和内置模板引擎的解析规则冲突的地方都可以使用 `literal` 标签处理。

模板注释

模板支持注释功能，该注释文字在最终页面不会显示，仅供模板制作人员参考和识别。

单行注释

格式：

```
{/* 注释内容 */ } 或 {// 注释内容 }
```

例如：

```
{// 这是模板注释内容 }
```

> 注意`{`和注释标记之间不能有空格。

多行注释

支持多行注释，例如：

```
{/* 这是模板
注释内容*/ }
```

模板注释支持多行，模板注释在生成编译缓存文件后会自动删除，这一点和Html的注释不同。

模板布局

ThinkPHP的模板引擎内置了布局模板功能支持，可以方便的实现模板布局以及布局嵌套功能。

有三种布局模板的支持方式：

第一种方式：全局配置方式

这种方式仅需在项目配置文件中添加相关的布局模板配置，就可以简单实现模板布局功能，比较适用于全站使用相同布局的情况，需要配置开启 `layout_on` 参数（默认不开启），并且设置布局入口文件名 `layout_name`（默认为layout）。

```
return  [
    'layout_on'     =>  true,
    'layout_name'   =>  'layout',
]
```

开启 `layout_on` 后，我们的模板渲染流程就有所变化，例如：

```
namespace app\index\controller;

use think\Controller;

Class User extends Controller
{
     public function add()
     {
         return $this->fetch('add');
     }
}
```

在不开启 `layout_on` 布局模板之前，会直接渲染 `application/index/view/user/add.html` 模板文件,开启之后，首先会渲染 `application/index/view/layout.html` 模板，布局模板的写法和其他模板的写法类似，本身也可以支持所有的模板标签以及包含文件，区别在于有一个特定的输出替换变量 `{__CONTENT__}`，例如，下面是一个典型的layout.html模板的写法：

```
{include file="public/header" /}
 {__CONTENT__}
{include file="public/footer" /}
```

读取layout模板之后，会再解析`user/add.html`模板文件，并把解析后的内容替换到layout布局模板文件的`{CONTENT}`特定字符串。

当然可以通过设置来改变这个特定的替换字符串，例如：

```
return [
    'layout_on'     =>  true,
    'layout_name'   =>  'layout',
    'layout_item'   =>  '{__REPLACE__}'
]
```

> 一个布局模板同时只能有一个特定替换字符串。

采用这种布局方式的情况下，一旦user/add.html 模板文件或者layout.html布局模板文件发生修改，都会导致模板重新编译。

如果需要指定其他位置的布局模板，可以使用：

```
return [
    'layout_on'     =>  true,
    'layout_name'   =>  'layout/layoutname',
    'layout_item'   =>  '{__REPLACE__}'
]
```

就表示采用`application/index/view/layout/layoutname.html`作为布局模板。

如果某些页面不需要使用布局模板功能，可以在模板文件开头加上`{__NOLAYOUT__}`字符串。

如果上面的user/add.html 模板文件里面包含有`{__NOLAYOUT__}`，则即使当前开启布局模板，也不会进行布局模板解析。

第二种方式：模板标签方式

这种布局模板不需要在配置文件中设置任何参数，也不需要开启`layout_on`，直接在模板文件中指定布局模板即可，相关的布局模板调整也在模板中进行。

以前面的输出模板为例，这种方式的入口还是在user/add.html 模板，但是我们可以修改下add模板文件的内容，在头部增加下面的布局标签（记得首先关闭前面的`layout_on`设置，否则可能出现布局循环）：

```
{layout name="layout" /}
```

表示当前模板文件需要使用`layout.html`布局模板文件，而布局模板文件的写法和上面第一种方式是一样的。当渲染`user/add.html`模板文件的时候，如果读取到layout标签，则会把当前模板的解析内容替换到layout布局模板的`{CONTENT}`特定字符串。

一个模板文件中只能使用一个布局模板，如果模板文件中没有使用任何layout标签则表示当前模板不使用任何布局。

如果需要使用其他的布局模板，可以改变layout的name属性，例如：

```
{layout name="newlayout" /}
```

还可以在layout标签里面指定要替换的特定字符串：

```
{layout name="Layout/newlayout" replace="[__REPLACE__]" /}
```

第三种方式：动态方法布局

使用内置的layout方法可以更灵活的在程序中控制模板输出的布局功能，尤其适用于局部需要布局或者关闭布局的情况，这种方式也不需要在配置文件中开启`layout_on`。例如：

```
namespace app\index\controller;

use think\Controller;

class User extends Controller
{
     public function add()
     {
         $this->view->engine->layout(true);
         return $this->fetch('add');
     }
}
```

表示当前的模板输出启用了布局模板，并且采用默认的layout布局模板。

如果当前输出需要使用不同的布局模板，可以动态的指定布局模板名称，例如：

```
namespace app\index\controller;

use think\Controller;
```

```
class User extends Controller
{
     public function add()
     {
         $this->view->engine->layout('Layout/newlayout');
         return $this->display('add');
     }
}
```

或者使用layout方法动态关闭当前模板的布局功能（这种用法可以配合第一种布局方式，例如全局配置已经开启了布局，可以在某个页面单独关闭）：

```
namespace app\index\controller;

use think\Controller;

class User extends Controller
{
     public function add()
     {
        // 临时关闭当前模板的布局功能
         $this->view->engine->layout(false);
         return $this->display('add');
     }
}
```

> 三种模板布局方式中，第一种和第三种是在程序中配置实现模板布局，第二种方式则是单纯通过模板标签在模板中使用布局。具体选择什么方式，需要根据项目的实际情况来了。

模板继承

模板继承是一项更加灵活的模板布局方式，模板继承不同于模板布局，甚至来说，应该在模板布局的上层。模板继承其实并不难理解，就好比类的继承一样，模板也可以定义一个基础模板（或者是布局），并且其中定义相关的区块（block），然后继承（extend）该基础模板的子模板中就可以对基础模板中定义的区块进行重载。

因此，模板继承的优势其实是设计基础模板中的区块（block）和子模板中替换这些区块。

每个区块由`{block} {/block}`标签组成。 下面就是基础模板中的一个典型的区块设计（用于设计网站标题）：

```
{block name="title"}<title>网站标题</title>{/block}
```

block标签必须指定name属性来标识当前区块的名称，这个标识在当前模板中应该是唯一的，block标签中可以包含任何模板内容，包括其他标签和变量，例如：

```
{block name="title"}<title>{$web_title}</title>{/block}
```

你甚至还可以在区块中加载外部文件：

```
{block name="include"}{include file="Public:header" /}{/block}
```

一个模板中可以定义任意多个名称标识不重复的区块，例如下面定义了一个`base.html`基础模板：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>{block name="title"}标题{/block}</title>
</head>
<body>
{block name="menu"}菜单{/block}
{block name="left"}左边分栏{/block}
{block name="main"}主内容{/block}
{block name="right"}右边分栏{/block}
{block name="footer"}底部{/block}
</body>
</html>
```

然后我们在子模板（其实是当前操作的入口模板）中使用继承：

```
{extend name="base" /}

{block name="title"}{$title}{/block}

{block name="menu"}
<a href="/" >首页</a>
<a href="/info/" >资讯</a>
<a href="/bbs/" >论坛</a>
{/block}

{block name="left"}{/block}

{block name="main"}
{volist name="list" id="vo"}
<a href="/new/{$vo.id}">{$vo.title}</a><br/>
 {$vo.content}
{/volist}
{/block}

{block name="right"}
 最新资讯：
{volist name="news" id="new"}
<a href="/new/{$new.id}">{$new.title}</a><br/>
{/volist}
{/block}

{block name="footer"}
{__block__}
 @ThinkPHP 版权所有
{/block}
```

上例中，我们可以看到在子模板中使用了extend标签来继承了base模板。
在子模板中，可以对基础模板中的区块进行重载定义，如果没有重新定义的话，则表示沿用基础模板中的区块定义，如果定义了一个空的区块，则表示删除基础模板中的该区块内容。上面的例子，我们就把left区块的内容删除了，其他的区块都进行了重载。而

```
{block name="footer"}
{__block__}@ThinkPHP 版权所有
{/block}
```

这一区块中有`{__block__}`这个标签，当区块中有这个标记时，就不只是直接重载这个区块，它表示引用所继承模板对应区块的内容到这个位置，最终这个区块是合并后的内容。所以这里footer区块最后的内容是： 底部@ThinkPHP 版权所有

extend标签的用法和include标签一样，你也可以加载其他模板：

```
{extend name="Public:base" /}
```

或者使用绝对文件路径加载

```
{extend name="./Template/Public/base.html" /}
```

在当前子模板中，只能定义区块而不能定义其他的模板内容，否则将会直接忽略，并且只能定义基础模板中已经定义的区块。

例如，如果采用下面的定义：

```
{block name="title"}<title>{$title}</title>{/block}
<a href="/" >首页</a>
<a href="/info/" >资讯</a>
<a href="/bbs/" >论坛</a>
```

导航部分将是无效的，不会显示在模板中。

模板可以多级继承，比如B继承了A，而C又继承了B，最终C中的区块会覆盖B和A中的同名区块，但C和B中的区块必须是A中已定义过的。

子模板中的区块定义顺序是随意的，模板继承的用法关键在于基础模板如何布局和设计规划了，如果结合原来的布局功能，则会更加灵活。

包含文件

在当前模版文件中包含其他的模版文件使用include标签，标签用法：

```
{include file='模版文件1,模版文件2,...' /}
```

> 包含的模板文件中不能再使用模板布局或者模板继承。

使用模版表达式

模版表达式的定义规则为：模块@控制器/操作

例如：

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

可以一次包含多个模版，例如：

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

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

使用模版文件

可以直接包含一个模版文件名（包含完整路径），例如：

```
{include file="../application/view/default/public/header.html" /}
```

> 路径以 项目目录/public/ 路径下为起点

传入参数

无论你使用什么方式包含外部模板，Include标签支持在包含文件的同时传入参数，例如，下

面的例子我们在包含header模板的时候传入了`title`和`keywords`参数：

```
{include file="Public/header" title="$title" keywords="开源WEB开发框架" /}
```

就可以在包含的header.html文件里面使用title和keywords变量，如下：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>[title]</title>
<meta name="keywords" content="[keywords]" />
</head>
```

上面title参数传入的是个变量`$title`，模板内的`[title]`最终会替换成`$title`的值，当然`$title`这个变量必须要存在。

> 包含文件中可以再使用include标签包含别的文件，但注意不要形成A包含A，或者A包含B而B又包含A这样的死循环。

> 由于模板解析的特点，从入口模板开始解析，如果外部模板有所更改，模板引擎并不会重新编译模板，除非在调试模式下或者缓存已经过期。如果部署模式下修改了包含的外部模板文件后，需要把模块的缓存目录清空，否则无法生效。

输出替换

模板输出替换

支持对模板文件输出的内容进行字符替换，定义后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。

> 5.1系统没有任何内置的替换规则。

如果需要全局替换的话，可以直接在 `template.php` 配置文件中添加：

```
'tpl_replace_string'  =>  [
    '__STATIC__'=>'/static',
        '__JS__' => '/static/javascript',
]
```

> 替换规则严格区分大小写

标签库

内置的模板引擎除了支持普通变量的输出之外，更强大的地方在于标签库功能。

> 标签库类似于Java的Struts中的JSP标签库，每一个标签库是一个独立的标签库文件，标签库中的每一个标签完成某个功能，采用XML标签方式（包括开放标签和闭合标签）。

标签库分为内置和扩展标签库，内置标签库是`Cx`标签库。

导入标签库

使用taglib标签导入当前模板中需要使用的标签库，例如：

```
{taglib name="html" /}
```

> 如果没有定义html标签库的话，则导入无效。

也可以导入多个标签库，使用：

```
{taglib name="html,article" /}
```

导入标签库后，就可以使用标签库中定义的标签了，假设article标签库中定义了read标签：

```
{article:read name="hello" id="data" }
{$data.id}:{$data.title}
{/article:read}
```

在上面的标签中，`{article:read}... {/article:read}`就是闭合标签，起始和结束标签必须成对出现。

如果是`{article:read name="hello" /}`就是开放标签。

> 闭合和开放标签取决于标签库中的定义，一旦定义后就不能混淆使用，否则就会出现错误。

内置标签

内置标签库无需导入即可使用，并且不需要加XML中的标签库前缀，ThinkPHP内置的标签库是Cx标签库，所以，Cx标签库中的所有标签，我们可以在模板文件中直接使用，我们可以这样使用：

```
{eq name="status" value="1" }
正常
{/eq}
```

如果Cx不是内置标签的话，可能就需要这么使用了：

```
{cx:eq name="status" value="1" }
正常
{/cx:eq}
```

更多的Cx标签库中的标签用法，参考后面的内置标签。

内置标签库可以简化模板中标签的使用，所以，我们还可以把其他的标签库定义为内置标签库（前提是多个标签库没有标签冲突的情况），例如：

```
'taglib_build_in'    =>    'cx,article'
```

配置后，上面的标签用法就可以改为：

```
{read name="hello" id="data" }
{$data.id}:{$data.title}
{/read}
```

标签库预加载

标签库预加载是指无需手动在模板文件中导入标签库即可使用标签库中的标签，通常用于某个标签库需要被大多数模板使用的情况。

在应用或者模块的配置文件中添加：

```
'taglib_pre_load'    =>    'article,html'
```

设置后，模板文件就不再需要使用

```
{taglib name="html,article" /}
```

但是仍然可以在模板中调用：

```
{article:read name="hello" id="data" }
{$data.id}:{$data.title}
{/article:read}
```

内置标签

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

内置标签主要包括：

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

循环标签

FOREACH 标签

foreach 标签的用法和PHP语法非常接近，用于循环输出数组或者对象的属性，用法如下：

```
$list = User::all();
$this->assign('list',$list);
```

模板文件中可以这样输出

```
{foreach $list as $key=>$vo }
    {$vo.id}:{$vo.name}
{/foreach}
```

可以不通过模板变量赋值，支持使用函数或者方法获取数据循环输出：

```
{foreach :model('user')->all() as $key=>$vo }
    {$vo.id}:{$vo.name}
{/foreach}
```

VOLIST 标签

volist 标签通常用于查询数据集或者二维数组的结果输出。 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}
```

可以直接使用函数设定数据集，而不需要在控制器中给模板变量赋值传入数据集变量，如：

```
{volist name=":model('user')->all()" id="vo"}
{$vo.name}
{/volist}
```

支持输出查询结果中的部分数据，例如输出其中的第5～15条记录

```
{volist name="list" id="vo" offset="5" length='10'}
{$vo.name}
{/volist}
```

输出偶数记录

```
{volist name="list" id="vo" mod="2" }
{eq name="mod" value="1"}{$vo.name}{/eq}
{/volist}
```

`mod` 属性还用于控制一定记录的换行，例如：

```
{volist name="list" id="vo" mod="5" }
{$vo.name}
{eq name="mod" value="4"}<br/>{/eq}
{/volist}
```

为空的时候输出提示：

```
{volist name="list" id="vo" empty="暂时没有数据" }
{$vo.id}|{$vo.name}
{/volist}
```

`empty` 属性不支持直接传入html语法，但可以支持变量输出，例如：

```
$this->assign('empty','<span class="empty">没有数据</span>');
$this->assign('list',$list);
```

然后在模板中使用：

```
{volist name="list" id="vo" empty="$empty" }
{$vo.id}|{$vo.name}
{/volist}
```

输出循环变量：

```
{volist name="list" id="vo" key="k" }
{$k}.{$vo.name}
{/volist}
```

如果没有指定 `key` 属性的话，默认使用循环变量i，例如：

```
{volist name="list" id="vo"  }
{$i}.{$vo.name}
{/volist}
```

如果要输出数组的索引，可以直接使用 `key` 变量，和循环变量不同的是，这个 `key` 是由数据本身决定，而不是循环控制的，例如：

```
{volist name="list" id="vo"  }
{$key}.{$vo.name}
{/volist}
```

`FOR` 标签

用法：

```
{for start="开始值" end="结束值" comparison="" step="步进值" name="循环变量名"
 }
{/for}
```

开始值、结束值、步进值和循环变量都可以支持变量，开始值和结束值是必须，其他是可选。comparison 的默认值是lt，name的默认值是i，步进值的默认值是1，举例如下：

```
{for start="1" end="100"}
{$i}
{/for}
```

解析后的代码是

```
for ($i=1;$i<100;$i+=1){
    echo $i;
}
```

比较标签

比较标签用于简单的变量比较，复杂的判断条件可以用if标签替换，比较标签是一组标签的集合，基本上用法都一致，如下：

```
{比较标签 name="变量" value="值"}
内容
{/比较标签}
```

系统支持的比较标签以及所表示的含义分别是：

标签	含义
eq或者 equal	等于
neq 或者notequal	不等于
gt	大于
egt	大于等于
lt	小于
elt	小于等于
heq	恒等于
nheq	不恒等于

他们的用法基本是一致的，区别在于判断的条件不同，并且所有的比较标签都可以和else标签一起使用。

例如，要求name变量的值等于value就输出，可以使用：

```
{eq name="name" value="value"}value{/eq}
```

或者

```
{equal name="name" value="value"}value{/equal}
```

也可以支持和else标签混合使用：

```
{eq name="name" value="value"}
相等
```

```
{else/}
不相等
{/eq}
```

当 name变量的值大于5就输出

```
{gt name="name" value="5"}value{/gt}
```

当name变量的值不小于5就输出

```
{egt name="name" value="5"}value{/egt}
```

比较标签中的变量可以支持对象的属性或者数组，甚至可以是系统变量，例如： 当vo对象的属性（或者数组，或者自动判断）等于5就输出

```
{eq name="vo.name" value="5"}
{$vo.name}
{/eq}
```

当vo对象的属性等于5就输出

```
{eq name="vo:name" value="5"}
{$vo.name}
{/eq}
```

当$vo['name']等于5就输出

```
{eq name="vo['name']" value="5"}
{$vo.name}
{/eq}
```

而且还可以支持对变量使用函数 当vo对象的属性值的字符串长度等于5就输出

```
{eq name="vo:name|strlen" value="5"}{$vo.name}{/eq}
```

变量名可以支持系统变量的方式，例如：

```
{eq name="Think.get.name" value="value"}相等{else/}不相等{/eq}
```

通常比较标签的值是一个字符串或者数字，如果需要使用变量，只需要在前面添加“$”标志： 当vo对象的属性等于$a就输出

```
{eq name="vo:name" value="$a"}{$vo.name}{/eq}
```

所有的比较标签可以统一使用compare标签（其实所有的比较标签都是compare标签的别名），例如： 当name变量的值等于5就输出

```
{compare name="name" value="5" type="eq"}value{/compare}
```

等效于

```
{eq name="name" value="5" }value{/eq}
```

其中type属性的值就是上面列出的比较标签名称

条件判断

SWITCH标签

用法：

```
{switch 变量 }
    {case value1 }输出内容1{/case}
    {case value2}输出内容2{/case}
    {default /}默认情况
{/switch}
```

使用示例：

```
{switch User.level}
    {case 1}value1{/case}
    {case 2}value2{/case}
    {default /}default
{/switch}
```

可以使用函数以及系统变量，例如：

```
{switch User.level|intval }
    {case 1}admin{/case}
    {default /}default
{/switch}
```

对于case属性可以支持多个条件的判断，使用”|”进行分割，例如：

```
{switch Think.get.type}
    {case gif|png|jpg}图像格式{/case}
    {default /}其他格式
{/switch}
```

表示如果 `$_GET["type"]` 是gif、png或者jpg的话，就判断为图像格式。

也可以对case的value属性使用变量，例如：

```
{switch $User.userId}
    {case $adminId}admin{/case}
    {case $memberId}member{/case}
```

```
{/switch}
```

> 使用变量方式的情况下，不再支持 | 分割的多个条件判断用法。

IF标签

用法：

```
{if 表达式}value1
{elseif 表达式 /}value2
{else /}value3
{/if}
```

用法示例：

```
{if ( $name == 1) OR ( $name > 100) } value1
{elseif $name == 2 /}value2
{else /} value3
{/if}
```

可以使用php代码，例如：

```
{if strtoupper($user['name']) == 'THINKPHP' }ThinkPHP
{else /} other Framework
{/if}
```

判断条件可以支持点语法和对象语法，例如：

```
{if $user.name == 'ThinkPHP'}ThinkPHP
{else /} other Framework
{/if}

{if $user->name == 'ThinkPHP'}ThinkPHP
{else /} other Framework
{/if}
```

> 如果某些特殊的要求下面，IF标签仍然无法满足要求的话，可以使用原生php代码或者PHP标签来直接书写代码。

范围判断

范围判断标签包括 `in` / `notin` / `between` / `notbetween` 四个标签，都用于判断变量是否中某个范围。

IN和NOTIN

用法： 假设我们中控制器中给id赋值为1：

```
$id =    1;
$this->assign('id',$id);
```

我们可以使用in标签来判断模板变量是否在某个范围内，例如：

```
{in name="id" value="1,2,3"}
id在范围内
{/in}
```

最后会输出：`id在范围内`。

如果判断不在某个范围内，可以使用notin标签：

```
{notin name="id" value="1,2,3"}
id不在范围内
{/notin}
```

最后会输出：`id不在范围内`。

可以把上面两个标签合并成为：

```
{in name="id" value="1,2,3"}
id在范围内
{else/}
id不在范围内
{/in}
```

name属性还可以支持直接判断系统变量，例如：

```
{in name="Think.get.id" value="1,2,3"}
$_GET['id'] 在范围内
{/in}
```

> 更多的系统变量用法可以参考系统变量部分。

value属性也可以使用变量，例如：

```
{in name="id" value="$range"}
id在范围内
{/in}
```

$range变量可以是数组，也可以是以逗号分隔的字符串。

value属性还可以使用系统变量，例如：

```
{in name="id" value="$Think.post.ids"}
id在范围内
{/in}
```

BETWEEN 和 NOTBETWEEN

可以使用 `between` 标签来判断变量是否在某个区间范围内，可以使用：

```
{between name="id" value="1,10"}
输出内容1
{/between}
```

同样，也可以使用 `notbetween` 标签来判断变量不在某个范围内：

```
{notbetween name="id" value="1,10"}
输出内容2
{/notbetween}
```

也可以使用 `else` 标签把两个用法合并，例如：

```
{between name="id" value="1,10"}
输出内容1
{else/}
输出内容2
{/between}
```

当使用 `between` 标签的时候， `value` 只需要一个区间范围，也就是只支持两个值，后面的值无效，例如

```
{between name="id" value="1,3,10"}
输出内容1
```

```
{/between}
```

实际判断的范围区间是`1~3`，而不是`1~10`，也可以支持字符串判断，例如：

```
{between name="id" value="A,Z"}
输出内容1
{/between}
```

name属性可以直接使用系统变量，例如：

```
{between name="Think.post.id" value="1,5"}
输出内容1
{/between}
```

value属性也可以使用变量，例如：

```
{between name="id" value="$range"}
输出内容1
{/between}
```

变量的值可以是字符串或者数组，还可以支持系统变量。

```
{between name="id" value="$Think.get.range"}
输出内容1
{/between}
```

PRESENT/NOTPRESENT标签

present标签用于判断某个变量是否已经定义，用法：

```
{present name="name"}
name已经赋值
{/present}
```

如果判断没有赋值，可以使用：

```
{notpresent name="name"}
name还没有赋值
{/notpresent}
```

可以把上面两个标签合并成为：

```
{present name="name"}
name已经赋值
{else /}
name还没有赋值
{/present}
```

present标签的name属性可以直接使用系统变量，例如：

```
{present name="Think.get.name"}
$_GET['name']已经赋值
{/present}
```

EMPTY/NOTEMPTY 标签

empty标签用于判断某个变量是否为空，用法：

```
{empty name="name"}
name为空值
{/empty}
```

如果判断没有赋值，可以使用：

```
{notempty name="name"}
name不为空
{/notempty}
```

可以把上面两个标签合并成为：

```
{empty name="name"}
name为空
{else /}
name不为空
{/empty}
```

name属性可以直接使用系统变量，例如：

```
{empty name="Think.get.name"}
$_GET['name']为空值
{/empty}
```

DEFINED 标签

DEFINED标签用于判断某个常量是否有定义，用法如下：

```
{defined name="NAME"}
NAME常量已经定义
{/defined}
```

> name属性的值要注意严格大小写

如果判断没有被定义，可以使用：

```
{notdefined name="NAME"}
NAME常量未定义
{/notdefined}
```

可以把上面两个标签合并成为：

```
{defined name="NAME"}
NAME常量已经定义
{else /}
NAME常量未定义
{/defined}
```

资源文件加载

传统方式的导入外部 `JS` 和 `CSS` 文件的方法是直接在模板文件使用：

```
<script type='text/javascript' src='/static/js/common.js'>
<link rel="stylesheet" type="text/css" href="/static/css/style.css" />
```

系统提供了专门的标签来简化上面的导入：

```
{load href="/static/js/common.js" /}
{load href="/static/css/style.css" /}
```

并且支持同时加载多个资源文件，例如：

```
{load href="/static/js/common.js,/static/css/style.css" /}
```

系统还提供了两个标签别名 `js` 和 `css` 用法和 `load` 一致，例如：

```
{js href="/static/js/common.js" /}
{css href="/static/css/style.css" /}
```

标签嵌套

模板引擎支持标签的多层嵌套功能，可以对标签库的标签指定可以嵌套。

系统内置的标签中，volist、switch、if、elseif、else、foreach、compare（包括所有的比较标签）、（not）present、（not）empty、（not）defined等标签都可以嵌套使用。例如：

```
{volist name="list" id="vo"}
    {volist name="vo['sub']" id="sub"}
        {$sub.name}
    {/volist}
{/volist}
```

上面的标签可以用于输出双重循环。

原生PHP

Php代码可以和标签在模板文件中混合使用，可以在模板文件里面书写任意的PHP语句代码，包括下面两种方式：

使用php标签

例如：

```
{php}echo 'Hello,world!';{/php}
```

我们建议需要使用PHP代码的时候尽量采用php标签，因为原生的PHP语法可能会被配置禁用而导致解析错误。

使用原生php代码

```
<?php echo 'Hello,world!'; ?>
```

注意：php标签或者php代码里面就不能再使用标签（包括普通标签和XML标签）了，因此下面的几种方式都是无效的：

```
{php}{eq name='name'value='value'}value{/eq}{/php}
```

Php标签里面使用了`eq`标签，因此无效

```
{php}if( {$user} != 'ThinkPHP' ) echo  'ThinkPHP' ;{/php}
```

Php标签里面使用了`{$user}`普通标签输出变量 ，因此无效。

```
{php}if( $user.name != 'ThinkPHP' ) echo  'ThinkPHP' ;{/php}
```

Php标签里面使用了`$user.name` 点语法变量输出 ，因此无效。

> 简而言之，在PHP标签里面不能再使用PHP本身不支持的代码。

如果设置了`tpl_deny_php`参数为true，就不能在模板中使用原生的PHP代码，但是仍然

支持PHP标签输出。

定义标签

ASSIGN标签

ASSIGN标签用于在模板文件中定义变量，用法如下：

```
{assign name="var" value="123" /}
```

在运行模板的时候，赋值了一个`var`的变量，值是`123`。

name属性支持系统变量，例如：

```
{assign name="Think.get.id" value="123" /}
```

表示在模板中给`$_GET['id']`赋值了`123`

value属性也支持变量，例如：

```
{assign name="var" value="$val" /}
```

或者直接把系统变量赋值给var变量，例如：

```
{assign name="var" value="$Think.get.name" /}
```

相当于，执行了：`$var = $_GET['name'];`

DEFINE标签

DEFINE标签用于中模板中定义常量，用法如下：

```
{define name="MY_DEFINE_NAME" value="3" /}
```

在运行模板的时候，就会定义一个`MY_DEFINE_NAME`的常量。

value属性可以支持变量（包括系统变量），例如：

```
{define name="MY_DEFINE_NAME" value="$name" /}
```

或者

```
{define name="MY_DEFINE_NAME" value="$Think.get.name" /}
```

标签扩展

错误和日志

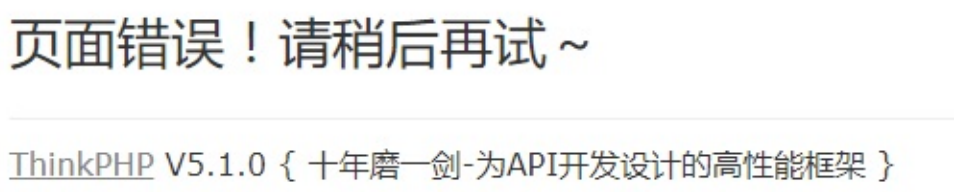
异常处理

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。

异常显示

在调试模式下，系统默认展示的异常页面：

> 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：



你可以通过设置 `exception_tmpl` 配置参数来自定义你的异常页面模板，默认的异常模板位于：

```
thinkphp/tpl/think_exception.tpl
```

你可以在应用配置文件`app.php`中更改异常模板

```
// 自定义异常页面的模板文件
'exception_tmpl'         => Env::get('app_path') . 'template/exception.tp
l',
```

默认的异常页面会返回`500`状态码，如果是一个`HttpException`异常则会返回HTTP的错误状态码。

异常忽略

> 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，可以使用下面方法设置。

默认框架会捕获所有的错误，包括PHP警告级别的错误，你可以在应用配置文件或者公共文件中设置要报告的错误级别，例如：

```
// 除了 E_NOTICE, 报告其他所有错误
error_reporting(E_ALL ^ E_NOTICE);
```

> 由于错误机制的注册顺序问题，在入口文件中设置错误级别无效。

> 系统产生的异常和错误都是程序的隐患，要尽早排除和解决，而不是掩盖。对于应用自己抛出的异常则做出相应的捕获处理。

异常处理接管

框架支持异常处理由开发者自定义类进行接管，需要在应用配置文件`app.php`中配置参数`exception_handle`。

最简单的方式是使用闭包定义异常处理，例如：

```
'exception_handle'       => function($e) {
  // 参数验证错误
  if ($e instanceof \think\exception\ValidateException) {
        return json($e->getError(), 422);
  }
```

```
  // 请求异常
  if ($e instanceof \think\exception\HttpException && request()->isAjax()
) {
        return response($e->getMessage(), $e->getStatusCode());
  }
},
```

如果需要更复杂的异常处理，可以把`exception_handle`配置定义为异常处理类的名称，例如：

```
    // 异常处理handle类 留空使用 \think\exception\Handle
    'exception_handle'       => '\\app\\common\\exception\\Http',
```

自定义类需要继承`think\exception\Handle`并且实现`render`方法，可以参考如下代码：

```
<?php
namespace app\common\exception;

use Exception;
use think\exception\Handle;
use think\exception\HttpException;
use think\exception\ValidateException;

class Http extends Handle
{
    public function render(Exception $e)
    {
        // 参数验证错误
        if ($e instanceof ValidateException) {
            return json($e->getError(), 422);
        }

        // 请求异常
        if ($e instanceof HttpException && request()->isAjax()) {
            return response($e->getMessage(), $e->getStatusCode());
        }

        // 其他错误交给系统处理
        return parent::render($e);
    }

}
```

自定义异常处理的主要作用是根据不同的异常类型发送不同的状态码和响应输出格式。

> 需要注意的是，如果自定义异常处理类没有再次调用系统 `render` 方法的话，配置 `http_exception_template` 就不再生效，具体可以参考 `Handle` 类内实现的功能。

手动抛出和捕获异常

ThinkPHP大部分情况异常都是自动抛出和捕获的，你也可以手动使用`throw`来抛出一个异常，例如：

```
// 使用think自带异常类抛出异常
throw new \think\Exception('异常消息', 10006);
```

系统提供了一个助手函数简化异常的代码，用法如下：

> exception('异常信息','异常代码','异常类')

异常代码默认为`0`，可以根据应用的异常设计来定义，异常类如果不传表示抛出默认的`think\Exception`异常，下面是示例：

```
// 使用助手函数抛出异常
exception('异常消息', 10006);
```

如果需要抛出自定义异常，可以使用：

```
// 抛出自定义异常
exception('异常消息', 10006,'\app\common\exception\NotFoundException');
```

手动捕获异常方式是使用`try-catch`，例如：

```
try {
    // 这里是主体代码
} catch (ValidateException $e) {
    // 这是进行验证异常捕获
    return json($e->getError());
} catch (\Exception $e) {
    // 这是进行异常捕获
    return json($e->getMessage());
}
```

> 支持使用`try-catch-finally`结构捕获异常。

可以使用PHP的异常捕获进行必要的处理，但需要注意一点，在异常捕获中不要使用`think\Controller`类的error、success和redirect方法，因为上述三个方法会抛出`HttpResponseException`异常，从而影响正常的异常捕获，例如：

```
try{
    Db::name('user')->find();
    $this->success('执行成功!');
}catch(\Exception $e){
    $this->error('执行错误');
}
```

应该改成

```
try{
    Db::name('user')->find();
}catch(\Exception $e){
    $this->error('执行错误');
}
$this->success('执行成功!');
```

HTTP 异常

可以使用 `\think\exception\HttpException` 类来抛出异常
框架提供了一个`abort`助手函数快速抛出一个HTTP异常：

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

use think\Controller;

class Index extends Controller
{
    public function index()
    {
      // 抛出 HTTP 异常
      throw new \think\exception\HttpException(404, '异常消息');
    }
}
```

系统提供了助手函数 `abort` 简化HTTP异常的处理，例如：
框架提供了一个`abort`助手函数快速抛出一个HTTP异常：

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

```
use think\Controller;

class Index extends Controller
{
    public function index()
    {
        // 抛出404异常
        abort(404, '页面异常');
    }
}
```

如果你的应用是API接口，那么请注意在客户端首先判断HTTP状态码是否正常，然后再进行数据处理，当遇到错误的状态码的话，应该根据状态码自行给出错误提示，或者采用下面的方法进行自定义异常处理。

部署模式下一旦抛出了`HttpException`异常，可以定义单独的异常页面模板，只需要在应用配置文件中增加：

```
'http_exception_template'    =>  [
    // 定义404错误的模板文件地址
    404 =>  Env::get('app_path') . '404.html',
    // 还可以定义其它的HTTP status
    401 =>  Env::get('app_path') . '401.html',
]
```

模板文件支持模板引擎中的标签。

> `http_exception_template`配置仅在部署模式下面生效。

日志处理

日志记录和写入由`\think\Log`类完成，通常我们使用`think\facade\Log`类进行静态调用。

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。

> 5.1的日志遵循 PSR-3 规范

日志配置

日志的配置文件是配置文件目录下的`log.php`文件，如果需要针对不同的模块设置不同的日志类型，则需要在模块配置目录下的`log.php`中配置，系统在进行日志写入之前会读取该配置文件进行初始化。

日志配置参数根据不同的日志类型有所区别，内置的日志类型包括：`file`、`socket`和`test`（仅用于测试，不会实际记录任何日志），日志类型使用`type`参数配置即可。

日志的全局配置参数包含（无论使用什么日志类型都支持）：

参数	描述
type	日志类型（或者驱动类名称）
level	允许记录的日志级别
allow_key	允许日志写入的授权key

> 默认的日志类型是 File 方式，可以通过驱动的方式来扩展支持更多的记录方式。

文件类型日志的话，还支持下列配置参数：

参数	描述
path	日志存储路径
file_size	日志文件大小限制（超出会生成多个文件）
apart_level	独立记录的日志级别
time_format	时间记录格式

> 为了避免同一个目录下面的日志文件过多的性能问题，日志文件会自动生成日期子目录。

下面是一个 `log.php` 配置示例：

```
return [
    // 日志记录方式，支持 file socket 或者自定义驱动类
    'type' => 'File',
    //日志保存目录
    'path' => '../logs/',
    //单个日志文件的大小限制，超过后会自动记录到第二个文件
    'file_size'      =>2097152,
    //日志的时间格式，默认是` c `
    'time_format'    =>'c'
],
```

> 系统并未提供关闭日志的方法，但有两种方式可以关闭日志的写入，第一种方式是设置日志类型为test，即不写入任何日志。第二种方式是设置日志记录级别，只记录需要的日志。

日志写入

手动记录

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

方法	描述
record()	记录日志信息到内存
save()	把保存在内存中的日志信息（用指定的记录方式）写入，并清空内存中的日志
write()	实时写入一条日志信息，会触发save操作

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');
```

> write方法同时也会把内存中的日志信息写入到文件。

> 为避免内存溢出，在命令行下面执行的话日志信息会实时写入。

日志级别

ThinkPHP对系统的日志按照级别来分类记录，按照 `PSR-3` 日志规范，日志的级别从低到高依次为： `debug` ， `info` ， `notice` ， `warning` ， `error` ， `critical` ， `alert` ， `emergency` ，ThinkPHP额外增加了一个 `sql` 日志级别仅用于记录 `SQL` 日志（并且仅当开启数据库调试模式有效）。

> 系统发生异常后记录的日志级别是 `error`

系统提供了不同日志级别的快速记录方法，例如：

```
Log::error('错误信息');
Log::info('日志信息');
// 和下面的用法等效
Log::record('错误信息','error');
Log::record('日志信息','info');
```

还封装了一个助手函数用于日志记录，例如：

```
trace('错误信息','error');
trace('日志信息','info');
```

也支持指定级别日志的输入，需要配置信息：

```
return [
    'type'  => 'File',
    // 日志记录级别，使用数组表示
    'level' => ['error','alert'],
],
```

上面的配置表示只记录`error`和`alert`级别的日志信息。

> 默认情况下是不会记录HTTP异常日志（避免受一些攻击的影响写入大量日志），除非你接管了系统的异常处理，重写了report方法。

上下文信息

日志可以传入上下文信息（数组），并且被替换到日志内容中，例如：

```
Log::info('日志信息{user}', ['user'=>'流年']);
```

实际写入日志的时候，`{user}`会被替换为流年。

独立日志

为了便于分析，`File`类型的日志还支持设置某些级别的日志信息单独文件记录，例如：

```
return [
    'type'          => 'file',
    // error和sql日志单独记录
    'apart_level'   =>  ['error','sql'],
],
```

设置后，就会单独生成`error`和`sql`两个类型的日志文件，主日志文件中将不再包含这两个级别的日志信息。

单文件日志

默认情况下，日志是按照日期为目录，按天为文件生成的，但如果希望仅生成单个文件（方

便其它的工具或者服务读取以及分析日志）。

```
return [
        'single'                =>      true,
    'file_size'         =>      1024*1024*10,
];
```

开启生成单个文件后，`file_size`和`apart_level`参数依然有效，超过文件大小限制后，系统会自动生成备份日志文件。

默认的单文件日志名是`single.log`，如果需要更改日志文件名，可以设置

```
return [
        'single'                =>      'single_file',
    'file_size'         =>      1024*1024*10,
];
```

那么实际生成的日志文件名是`single_file.log`，如果设置了`apart_level`的话，可能还会生成`single_file.error.log`之类的日志。

写入授权

日志支持写入授权，我们可以设置某个请求的日志授权Key，然后设置允许授权写入的配置`Key`，实现个别用户日志记录的功能，从而提高高负载下的日志记录性能。

首先需要在应用配置文件或者应用公共文件中添加当前访问的授权Key定义，例如：

```
// 设置IP为授权Key
Log::key(Request::ip());
```

然后在日志配置参数中增加`allow_key`参数，如下：

```
return  [
    // 日志类型为File
    'type'      =>  'File',
    // 授权只有202.12.36.89 才能记录日志
    'allow_key' =>  ['202.12.36.89'],
]
```

清空日志

一旦执行save或者write方法后，内存中的日志信息就会被自动清空，如果需要手动清空可以使用：

```
Log::clear();
```

在清空日志或者调用`save/write`方法之前，你可以使用getLog方法获取内存中的日志。

```
// 获取错误类型日志
$error = Log::getLog('error');
// 获取全部日志
$logs = Log::getLog();
```

> 日志清空仅仅是清空内存中的日志。

调试

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。

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

应用默认是部署模式，在开发阶段，可以修改应用配置文件`app.php`中的`app_debug`参数（或者环境变量`APP_DEBUG`）开启调试模式，上线部署后切换到部署模式。

```
// 开启调试模式
'app_debug' => true,
```

> 在模块配置文件中设置调试模式无效

除此之外，还可以在应用根目录下面定义`.env`文件，并且定义`APP_DEBUG`环境变量参数，这样便于在部署环境中设置环境变量来开启和关闭调试模式。

`.env`文件的定义格式如下：

```
// 设置开启调试模式
APP_DEBUG =  true
// 其它的环境变量设置
// ...
```

> 定义了`.env`文件后，配置文件中定义`app_debug`参数无效。

调试模式的优势在于：

- 开启日志记录，任何错误信息和调试信息都会详细记录，便于调试；
- 会详细记录整个执行过程；
- 模板修改可以即时生效；
- 记录SQL日志，方便分析SQL；
- 通过Trace功能更好的调试和发现错误；

- 发生异常的时候会显示详细的异常信息；

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

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，那么可以在`app.php`文件中如下设置：

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

Trace调试

调试模式并不能完全满足我们调试的需要，有时候我们需要手动的输出一些调试信息。除了本身可以借助一些开发工具进行调试外，ThinkPHP还提供了一些内置的调试工具和函数。

`Trace`调试功能就是ThinkPHP提供给开发人员的一个用于开发调试的辅助工具。可以实时显示当前页面或者请求的请求信息、运行情况、SQL执行、错误信息和调试信息等，并支持自定义显示，并且支持没有页面输出的操作调试。

> Trace调试功能对调试模式和部署模式都有效，可以单独开启和关闭。
> 只是在部署模式下面，显示的调试信息没有调试模式完整，通常我们建议Trace配合调试模式一起使用。

开启Trace调试

该功能默认关闭，要开启Trace调试功能，只需要在应用配置`app.php`文件中配置下面参数：

```
// 开启应用Trace调试
'app_trace' =>  true,
```

> 如果定义了环境变量 `APP_TRACE` ，那么以环境变量配置为准。

页面Trace功能开启后，运行后并且你的页面有输出的话，页面右下角会显示`ThinkPHP`的LOGO：

LOGO后面的数字就是当前页面的执行时间（单位是秒） 点击该图标后，会展开详细的Trace信息，如图：

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

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

Trace的选项卡是可以定制和扩展的，如果你希望增加新的选项卡，则可以修改配置目录下的`trace.php`文件中的配置参数如下：

```
return [
    'type'       =>  'Html',
    'trace_tabs' =>  [
         'base'=>'基本',
         'file'=>'文件',
         'info'=>'流程',
         'error'=>'错误',
         'sql'=>'SQL',
         'debug'=>'调试',
         'user'=>'用户'
     ]
];
```

> `base` 和 `file` 是系统内置的，其它的选项其实都属于日志的等级（user是用户自定义的日志等级）。

也可以把某几个选项卡合并，例如：

```
return [
    'type'        =>  'Html',
    'trace_tabs' =>  [
         'base'=>'基本',
         'file'=>'文件',
         'error|notice|warning'=>'错误',
         'sql'=>'SQL',
         'debug|info'=>'调试',
     ]
];
```

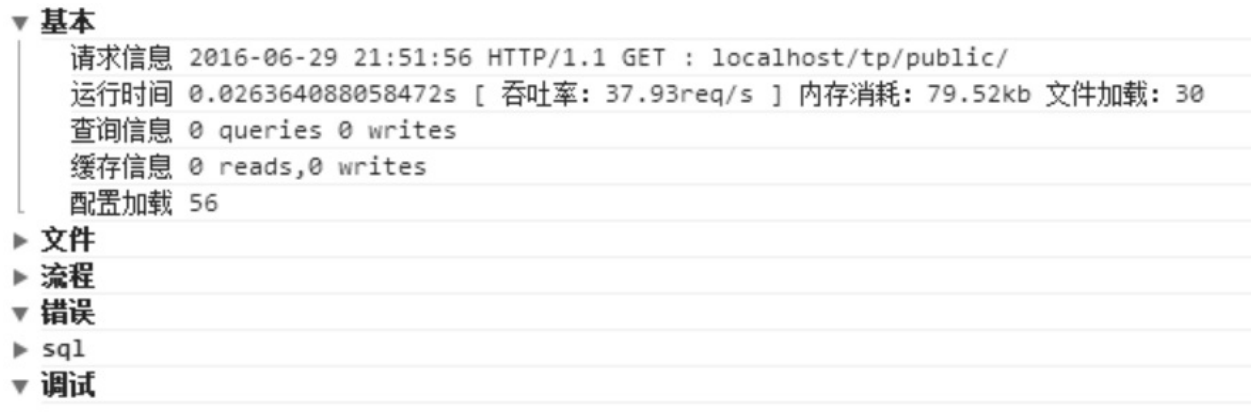
更改后的Trace显示效果如图：

浏览器控制台输出

trace功能支持在浏览器的`console`直接输出，这样可以方便没有页面输出的操作功能调试，只需要在配置文件中设置：

```
    // 使用浏览器console输出trace信息
    'type'  =>  'console',
```

运行后打开浏览器的console控制台可以看到如图所示的信息：



浏览器Trace输出同样支持`trace_tabs`设置。

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（实际使用`think\facade\Debug`类即可）可以很方便的获取某个区间的运行时间和内存占用情况。 例如：

```
Debug::remark('begin');
// ...其他代码段
Debug::remark('end');
// ...也许这里还有其他代码
// 进行统计区间
echo Debug::getRangeTime('begin','end').'s';
```

表示统计begin位置到end位置的执行时间（单位是秒），begin必须是一个已经标记过的位置，如果这个时候end位置还没被标记过，则会自动把当前位置标记为end标签，输出的结果类似于：`0.0056s`

默认的统计精度是小数点后4位，如果觉得这个统计精度不够，还可以设置例如：

```
echo Debug::getRangeTime('begin','end',6).'s';
```

可能的输出会变成：`0.005587s`

如果你的环境支持内存占用统计的话，还可以使用`getRangeMem`方法进行区间内存开销统计（单位为kb），例如：

```
echo Debug::getRangeMem('begin','end').'kb';
```

第三个参数使用m表示进行内存开销统计，输出的结果可能是：`625kb`

同样，如果end标签没有被标记的话，会自动把当前位置先标记位end标签。

助手函数

系统还提供了助手函数`debug`用于完成相同的作用，上面的代码可以改成：

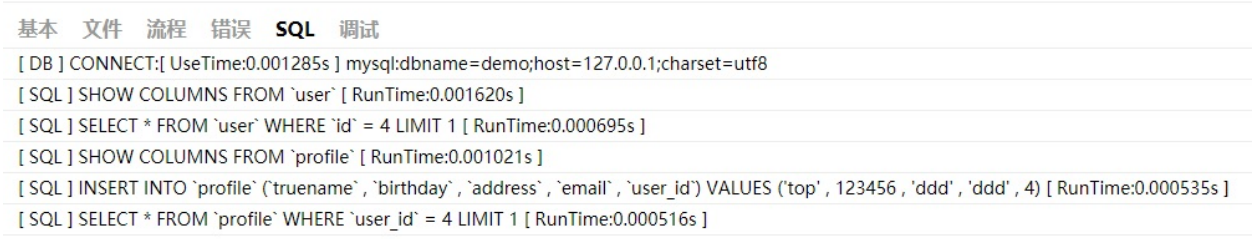
```
debug('begin');
// ...其他代码段
debug('end');
```

```
// ...也许这里还有其他代码
// 进行统计区间
echo debug('begin','end').'s';
echo debug('begin','end',6).'s';
echo debug('begin','end','m').'kb';
```

SQL调试

查看页面Trace

通过查看页面Trace信息可以看到当前请求所有执行的SQL语句，例如：



查看SQL日志

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（甚至包含性能分析）。

> 通常我们建议设置把SQL日志级别写入到单独的日志文件中，具体可以参考日志处理部分。

下面是一个典型的SQL日志：

```
[ SQL ] SHOW COLUMNS FROM `think_user` [ RunTime:0.001339s ]
[ SQL ] SELECT * FROM `think_user` LIMIT 1 [ RunTime:0.000539s ]
```

如果需要对SQL性能进行分析的话，可以在数据库配置文件中开启下面参数：

```
        // 是否需要进行SQL性能分析
        'sql_explain'     => false,
```

开启后，日志会记录类似下面的信息

```
[ SQL ] SHOW COLUMNS FROM `think_user` [ RunTime:0.001339s ]
[ EXPLAIN : array ( 'id' => '1', 'select_type' => 'SIMPLE', 'table' => 't
hink_user', 'partitions' => NULL, 'type' => 'ALL', 'possible_keys' => NUL
L, 'key' => NULL, 'key_len' => NULL, 'ref' => NULL, 'rows' => '82', 'filt
ered' => '100.00', 'extra' => NULL, ) ]
[ SQL ] SELECT * FROM `think_user` 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);
});
```

> 一旦使用了SQL侦听，默认就不会记录SQL日志了，需要自己接管操作。

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`

> `getLastSql` 方法只能获取最后执行的 SQL 记录。

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。

变量调试

除了Trace调试之外，系统还提供了`\think\Debug`类用于各种调试。

输出某个变量是开发过程中经常会用到的调试方法，除了使用php内置的`var_dump`和`print_r`之外，ThinkPHP框架内置了一个对浏览器友好的`dump`方法，用于输出变量的信息到浏览器查看。

用法：

```
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

如果第二个参数为`false`则返回要输出的字符串而不是直接输出。

使用示例：

```
 $blog = Db::name('blog')->where('id', 3)->find();
 dump($blog);
```

在浏览器输出的结果是：

```
array(12) {
   ["id"]           => string(1) "3"
   ["name"]         => string(0) ""
   ["user_id"]      => string(1) "0"
   ["cate_id"]      => string(1) "0"
   ["title"]        => string(4) "test"
   ["content"]      => string(4) "test"
   ["create_time"]  => string(1) "0"
   ["update_time"]  => string(1) "0"
   ["status"]       => string(1) "0"
   ["read_count"]   => string(1) "0"
   ["comment_count"] => string(1) "0"
   ["tags"]         => string(0) ""
```

```
}
```

如果需要在调试变量输出后中止程序的执行，可以使用`halt`函数，例如：

```
$blog = Db::name('blog')->where('id', 3)->find();

halt($blog);
echo '这里的信息是看不到的';
```

执行后会输出同样的结果并中止执行后续的程序。

> 5.1版本中为了方便查看，`dump`输出的模型对象实例显示为数组，如果你需要输出实际的模型属性可以使用`var_dump`方法。

远程调试

`ThinkPHP5.0`版本开始，提供了`Socket`日志驱动用于本地和远程调试。

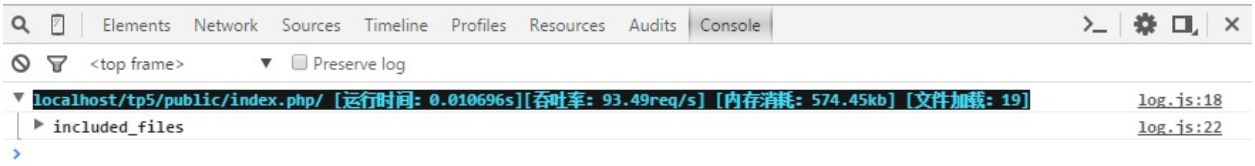
Socket调试

只需要在配置文件中设置如下：

```
return [
    'type'                  => 'socket',
    'host'                  => 'slog.thinkphp.cn',
    //日志强制记录到配置的client_id
    'force_client_ids'      => [],
    //限制允许读取日志的client_id
    'allow_client_ids'      => [],
]
```

> 上面的host配置地址是官方提供的公用服务端，首先需要去申请client_id 。

使用Chrome浏览器运行后，打开`审查元素->Console`，可以看到如下所示：



`SocketLog`通过`websocket`将调试日志打印到浏览器的`console`中。你还可以用它来分析开源程序，分析SQL性能，结合taint分析程序漏洞。

安装Chrome插件

`SocketLog`首先需要安装`chrome`插件，Chrome插件安装页面 （需翻墙）

使用方法

- 首先，请在chrome浏览器上安装好插件。
- 安装服务端`npm install -g socketlog-server`，运行命令`socketlog-server`即可启动服务。 将会在本地起一个websocket服务 ，监听端口是1229 。
- 如果想服务后台运行： `socketlog-server > /dev/null &`

参数

- `client_id`：在chrome浏览器中，可以设置插件的`Client_ID`，Client_ID是你任意指定的字符串。

- 设置`client_id`后能实现以下功能：

- 1，配置`allow_client_ids`配置项，让指定的浏览器才能获得日志，这样就可以把调试代码带上线。 普通用户访问不会触发调试，不会发送日志。 开发人员访问就能看的调试日志， 这样利于找线上bug。 Client_ID 建议设置为姓名拼音加上随机字符串，这样如果有员工离职可以将其对应的`client_id`从配置项`allow_client_ids`中移除。 `client_id`除了姓名拼音，加上随机字符串的目的，以防别人根据你公司员工姓名猜测出`client_id`,获取线上的调试日志。

- 设置`allow_client_ids`示例代码：

```
'allow_client_ids'=>['thinkphp_zfH5NbLn','luofei_DJq0z80H'],
```

- 2, 设置`force_client_ids`配置项，让后台脚本也能输出日志到chrome。 网站有可能用了队列，一些业务逻辑通过后台脚本处理， 如果后台脚本需要调试，你也可以将日志打印到浏览器的console中， 当然后台脚本不和浏览器接触，不知道当前触发程序的是哪个浏览器，所以我们需要强制将日志打印到指定`client_id`的浏览器上面。 我们在后台脚本中使用SocketLog时设置`force_client_ids`配置项指定要强制输出浏览器的`client_id`即可。

验证

验证器

ThinkPHP `5.1` 推荐使用验证器进行数据验证（也支持使用 `\think\Validate` 类进行独立验证）。

验证器定义

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',
    ];

}
```

可以直接在验证器类中使用 `message` 属性定义错误提示信息，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];
```

```
}
```

> 如果没有定义错误提示信息，则使用系统默认的提示信息

数据验证

在需要进行 `User` 验证的控制器方法中，添加如下代码即可：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        $data = [
            'name'  => 'thinkphp',
            'email' => 'thinkphp@qq.com',
        ];

        $validate = new \app\index\validate\User;

        if (!$validate->check($data)) {
            dump($validate->getError());
        }
    }
}
```

事实上控制器类提供了一个 `validate` 方法可以更方便的进行验证，如下：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    public function index()
    {
        $result = $this->validate(
            [
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ],
            'app\index\validate\User');

        if (true !== $result) {
            // 验证失败 输出错误信息
            dump($result);
```

```
        }
    }
}
```

批量验证

默认情况下，一旦有某个数据的验证规则不符合，就会停止后续数据及规则的验证，如果希望批量进行验证，可以设置：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    // 是否批量验证
    protected $batchValidate = true;

    public function index()
    {
        $result = $this->validate(
            [
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ],
            'app\index\validate\User');

        if (true !== $result) {
            // 验证失败 输出错误信息
            dump($result);
        }
    }
}
```

> 批量验证如果验证不通过，返回的是一个错误信息的数组。

抛出验证异常

默认情况下验证失败后不会抛出异常，如果希望验证失败自动抛出异常，可以在控制器类中添加设置：

```
namespace app\index\controller;

use think\Controller;

class Index extends Controller
{
    // 验证失败是否抛出异常
```

```
    protected $failException = true;

    public function index()
    {
        $result = $this->validate(
            [
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ],
            'app\index\validate\User');
    }
}
```

设置开启了验证失败后抛出异常的话，无需手动获取错误信息，会自动抛出`think\exception\ValidateException`异常或者自己捕获处理。

自定义验证规则

系统内置了一些常用的规则（参考后面的内置规则），如果不能满足需求，可以在验证器重添加额外的验证方法，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name'  =>  'checkName:thinkphp',
        'email' =>  'email',
    ];

    protected $message = [
        'name'  =>  '用户名必须',
        'email' =>  '邮箱格式错误',
    ];

    // 自定义验证规则
    protected function checkName($value,$rule,$data=[])
    {
        return $rule == $value ? true : '名称错误';
    }
}
```

验证方法可以传入的参数共有`5`个（后面三个根据情况选用），依次为：

- 验证数据
- 验证规则
- 全部数据（数组）

- 字段名
- 字段描述

> 自定义的验证规则方法名不能和已有的规则冲突。

验证规则

验证规则的定义通常有两种方式，如果你使用了验证器的话，通常通过 `rule` 属性定义验证规则，而如果使用的是独立验证的话，则是通过 `rule` 方法进行定义。

属性定义

属性定义方式仅限于验证器，通常类似于下面的方式：

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

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

}
```

> 系统内置了一些常用的验证规则可以满足大部分的验证需求，具体每个规则的含义参考内置规则一节。

因为一个字段可以使用多个验证规则（如上面的 `age` 字段定义了 `number` 和 `between` 两个验证规则），在一些特殊的情况下，为了避免混淆可以在 `rule` 属性中使用数组定义规则。

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

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => ['require', 'max' => 25, 'regex' => '/^[\w|\d]\w+/'],
      'age'   => ['number', 'between' => '1,120'],
      'email' => 'email',
    ];
```

```
}
```

方法定义

如果使用的是独立验证（即手动调用验证类进行验证）方式的话，通常使用`rule`方法进行验证规则的设置，举例说明如下。

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

$data = [
    'name'  => 'thinkphp',
    'email' => 'thinkphp@qq.com'
];

if (!$validate->check($data)) {
    dump($validate->getError());
}
```

> `rule`方法传入数组表示批量设置规则。

`rule`方法还可以支持使用对象化的规则定义，这是5.1版本新增的特性。

我们把上面的验证代码改为

```
use think\validate\ValidateRule as Rule;

$validate = new \think\Validate;
$validate->rule('age', Rule::isNumber()->between([1,120]))
->rule([
    'name'  => Rule::isRequire()->max(25),
    'email' => Rule::isEmail(),
]);

$data = [
    'name'  => 'thinkphp',
    'email' => 'thinkphp@qq.com'
];

if (!$validate->check($data)) {
    dump($validate->getError());
}
```

可以对某个字段使用闭包验证，例如：

```
$validate = new \think\Validate;
$validate->rule([
    'name'  => function($value) {
        return 'thinkphp' == strtolower($value) ? true : false;
    },
]);
```

闭包支持传入两个参数，第一个参数是当前字段的值（必须），第二个参数是所有数据（可选）。

> 如果使用了闭包进行验证，则不再支持对该字段使用多个验证规则。

闭包函数如果返回true则表示验证通过，返回false表示验证失败并使用系统的错误信息，如果返回字符串，则表示验证失败并且以返回值作为错误提示信息。

```
$validate = new \think\Validate;
$validate->rule([
    'name'  => function($value) {
        return 'thinkphp' == strtolower($value) ? true : '用户名错误';
    },
]);
```

> 属性方式定义验证规则不支持使用对象化规则定义和闭包定义

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：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

}
```

```
$data = [
    'name'  => 'thinkphp',
    'age'   => 121,
    'email' => 'thinkphp@qq.com',
];

$validate = new \app\index\validate\User;
$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：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
```

```
        'name'  => 'require|max:25',
        'age|年龄'   => 'number|between:1,120',
        'email' => 'email',
    ];

}
```

会输出 `年龄只能在 1 - 120 之间`。

单独定义提示信息

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：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message = [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄必须在1~120之间',
        'email'        => '邮箱格式错误',
    ];
}
```

```
$data = [
    'name'  => 'thinkphp',
    'age'   => 121,
    'email' => 'thinkphp@qq.com',
];

$validate = new \app\index\validate\User;
$result = $validate->check($data);

if(!$result){
    echo $validate->getError();
}
```

会输出 `年龄必须在1~120之间`。

使用多语言

5.1的验证信息提示支持多语言功能，你只需要给相关错误提示信息定义语言包，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
      'name'  => 'require|max:25',
      'age'   => 'number|between:1,120',
      'email' => 'email',
    ];

    protected $message = [
      'name.require' => 'name_require',
      'name.max'     => 'name_max',
      'age.number'   => 'age_number',
      'age.between'  => 'age_between',
      'email'        => 'email_error',
    ];
}
```

你可以在语言包文件中添加下列定义：

```
'name_require ' =>      '姓名必须',
'name_max'              =>      '姓名最大长度不超过25个字符',
'age_between'   =>      '年龄必须在1~120之间',
'age_number'    =>      '年龄必须是数字',
'email_error'   =>      '邮箱格式错误',
```

> 系统内置的验证错误提示均支持多语言（参考框架目录下的 lang/zh-cn.php 语言定义文件）。

验证场景

验证场景

验证器重支持定义场景，并且验证不同场景的数据，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];

    protected $scene = [
        'edit'  =>  ['name','age'],
    ];

}
```

然后可以在验证方法中制定验证的场景

```
$data = [
    'name'  => 'thinkphp',
    'age'   => 10,
    'email' => 'thinkphp@qq.com',
];

$result = $this->validate($data,'app\index\validate\User.edit');

if(true !== $result){
    // 验证失败 输出错误信息
    dump($result);
}
```

如果你直接调用验证器类的话直接使用 `scene` 方法指定验证场景。

```
namespace app\index\controller;

use app\index\validate\User as UserValidate;
use think\Controller;

class Index extends Controller
{
    public function index()
    {
        $data = [
            'name'  => 'thinkphp',
            'email' => 'thinkphp@qq.com',
        ];

        $validate = new UserValidate;

        if (!$validate->scene('edit')->check($data)) {
            dump($validate->getError());
        }
    }
}
```

可以单独为某个场景定义方法（方法的命名规范是`scene`+场景名），并且对某些字段的规则重新设置，例如：

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $rule =   [
        'name'  => 'require|max:25',
        'age'   => 'number|between:1,120',
        'email' => 'email',
    ];

    protected $message  =   [
        'name.require' => '名称必须',
        'name.max'     => '名称最多不能超过25个字符',
        'age.number'   => '年龄必须是数字',
        'age.between'  => '年龄只能在1-120之间',
        'email'        => '邮箱格式错误',
    ];

    // edit 验证场景定义
    public function sceneEdit()
    {
        return $this->only(['name','age'])
```

```
                ->append('name', 'min:5')
            ->remove('age', 'between')
            ->append('age', 'require|max:100');
    }
}
```

主要方法说明如下：

方法名	描述
only	场景需要验证的字段
remove	移除场景中的字段的部分验证规则
append	给场景中的字段需要追加验证规则

如果对同一个字段进行多次规则补充（包括移除和追加），必须使用下面的方式：

```
remove('field', ['rule1','rule2'])
// 或者
remove('field', 'rule1|rule2')
```

下面的方式会导致rule1规则remove不成功

```
remove('field', 'rule1')
->remove('field', 'rule2')
```

路由验证

可以在路由规则定义的时候调用 `validate` 方法指定验证器类对请求的数据进行验证。

例如下面的例子表示对请求数据使用验证器类 `app\index\validate\User` 进行自动验证，并且使用 `edit` 验证场景：

```
Route::post('hello/:id', 'index/hello')
        ->model('id', 'app\index\model\User')
        ->validate('app\index\validate\User','edit');
```

或者不使用验证器而直接传入验证规则

```
Route::post('hello/:id', 'index/hello')
    ->model('id', 'app\index\model\User')
    ->validate([
        'name'  =>      'min:5|max:50',
        'email' =>      'email',
    ]);
```

也支持使用对象化规则定义

```
Route::post('hello/:id', 'index/hello')
    ->model('id', 'app\index\model\User')
    ->validate([
        'name'  =>      ValidateRule::min(5)->max(50),
        'email' =>      ValidateRule::isEmail(),
    ]);
```

内置规则

系统内置了一些常用的验证规则，可以完成大部分场景的验证需求，包括：

- 格式验证类
- 长度和区间验证类
- 字段比较类
- filter验证
- 正则验证
- 上传验证
- 行为验证
- 其它验证

> 验证规则严格区分大小写

格式验证类

格式验证类在使用静态方法调用的时候支持两种方式调用（以 `number` 验证为例，可以使用 `number()` 或者 `isNumber()` ）。

require

验证某个字段必须，例如：

```
'name'=>'require'
```

> 如果验证规则没有添加 `require` 就表示没有值的话不进行验证

> 由于 `require` 属于PHP保留字，所以在使用方法验证的时候必须使用 `isRequire` 或者 `must` 方法调用。

number

验证某个字段的值是否为数字，例如：

```
'num'=>'number'
```

integer

验证某个字段的值是否为数字（采用 `filter_var` 验证），例如：

```
'num'=>'integer'
```

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 或者 bool

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'
```

chs

验证某个字段的值只能是汉字，例如：

```
'name'=>'chs'
```

chsAlpha

验证某个字段的值只能是汉字、字母，例如：

```
'name'=>'chsAlpha'
```

chsAlphaNum

验证某个字段的值只能是汉字、字母和数字，例如：

```
'name'=>'chsAlphaNum'
```

chsDash

验证某个字段的值只能是汉字、字母、数字和下划线_及破折号-，例如：

```
'name'=>'chsDash'
```

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'
```

mobile

验证某个字段的值是否为有效的手机，例如：

```
'mobile'=>'mobile'
```

idCard

验证某个字段的值是否为有效的身份证格式，例如：

```
'id_card'=>'idCard'
```

macAddr

验证某个字段的值是否为有效的MAC地址，例如：

```
'mac'=>'macAddr'
```

zip

验证某个字段的值是否为有效的邮政编码，例如：

```
'zip'=>'zip'
```

长度和区间验证类

in

验证某个字段的值是否在某个范围，例如：

```
'num'=>'in:1,2,3'
```

notIn

验证某个字段的值不在某个范围，例如：

```
'num'=>'notIn:1,2,3'
```

between

验证某个字段的值是否在某个区间，例如：

```
'num'=>'between:1,10'
```

notBetween

验证某个字段的值不在某个范围，例如：

```
'num'=>'notBetween:1,10'
```

length:num1,num2

验证某个字段的值的长度是否在某个范围，例如：

```
'name'=>'length:4,25'
```

或者指定长度

```
'name'=>'length:4'
```

> 如果验证的数据是数组，则判断数组的长度。
> 如果验证的数据是File对象，则判断文件的大小。

max:number

验证某个字段的值的最大长度，例如：

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

> 如果验证的数据是数组，则判断数组的长度。
> 如果验证的数据是File对象，则判断文件的大小。

min:number

验证某个字段的值的最小长度，例如：

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

> 如果验证的数据是数组，则判断数组的长度。
> 如果验证的数据是File对象，则判断文件的大小。

after:日期

验证某个字段的值是否在某个日期之后，例如：

```
'begin_time' => 'after:2016-3-18',
```

before:日期

验证某个字段的值是否在某个日期之前，例如：

```
'end_time'   => 'before:2016-10-01',
```

expire:开始时间,结束时间

验证当前操作（注意不是某个值）是否在某个有效日期之内，例如：

```
'expire_time'   => 'expire:2016-2-1,2016-10-01',
```

allowIp:allow1,allow2,...

验证当前请求的IP是否在某个范围，例如：

```
'name'   => 'allowIp:114.45.4.55',
```

该规则可以用于某个后台的访问权限，多个IP用逗号分隔

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',
```

多个IP用逗号分隔

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：

```
'repassword'=>'require|confirm:password'
```

支持字段自动匹配验证规则，如`password`和`password_confirm`是自动相互验证的，只需要使用

```
'password'=>'require|confirm'
```

会自动验证和`password_confirm`进行字段比较是否一致，反之亦然。

different

验证某个字段是否和另外一个字段的值不一致，例如：

```
'name'=>'require|different:account'
```

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'
```

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'
```

字段比较

验证对比其他字段大小（数值大小对比），例如：

```
'price'=>'lt:market_price'
'price'=>'<:market_price'
```

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，例如在验证器类中定义regex属性

```
namespace app\index\validate;

use think\Validate;

class User extends Validate
{
    protected $regex = [ 'zip' => '\d{6}'];

    protected $rule = [
        'name'  =>  'require|max:25',
        'email' =>  'email',
    ];

}
```

然后就可以使用

```
'zip'   =>      'regex:zip',
```

上传验证

file

验证是否是一个上传文件

image:width,height,type

验证是否是一个图像文件，width height和type都是可选，width和height必须同时定义。

fileExt:允许的文件后缀

验证上传文件后缀

fileMime:允许的文件类型

验证上传文件类型

fileSize:允许的文件字节大小

验证上传文件大小

行为验证

使用行为验证数据，例如：

```
'data'=>'behavior:\app\index\behavior\Check'
```

其它验证

unique:table,field,except,pk

验证当前请求的字段值是否为唯一的，例如：

```
// 表示验证name字段的值是否在user表（不包含前缀）中唯一
'name'   => 'unique:user',
// 验证其他字段
'name'   => 'unique:user,account',
// 排除某个主键值
'name'   => 'unique:user,account,10',
// 指定某个主键值排除
'name'   => 'unique:user,account,10,user_id',
```

如果需要对复杂的条件验证唯一，可以使用下面的方式：

```
// 多个字段验证唯一验证条件
'name'   => 'unique:user,status^account',
// 复杂验证条件
'name'   => 'unique:user,status=1&account='.$data['account'],
```

requireIf:field,value

验证某个字段的值等于某个值的时候必须，例如：

```
// 当account的值等于1的时候 password必须
'password'=>'requireIf:account,1'
```

requireWith:field

验证某个字段有值的时候必须，例如：

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

requireCallback:callable

验证当某个callable为真的时候字段必须，例如：

```
// 使用check_require方法检查是否需要验证age字段必须
'age'=>'requireCallback:check_require|number'
```

用于检查是否需要验证的方法支持两个参数，第一个参数是当前字段的值，第二个参数则是所有的数据。

```
function check_require($value, $data){
    if(empty($data['birthday'])){
        return true;
    }
}
```

只有check_require函数返回true的时候age字段是必须的，并且会进行后续的其它验证。

独立验证

创建验证

任何时候，都可以直接使用`think\Validate`类而不需要定义验证器类进行独立的验证操作，例如：

```
$validate = Validate::make([
    'name'  => 'require|max:25',
    'email' => 'email'
]);

$data = [
    'name'  => 'thinkphp',
    'email' => 'thinkphp@qq.com'
];

if (!$validate->check($data)) {
    dump($validate->getError());
}
```

`make`方法直接传入验证规则（数组），`check`方法传入需要验证的数据（数组）。

设置规则

也可以使用`rule`方法设置单个或者批量验证规则，例如：

```
Validate::make()
    ->rule('zip', '/^\d{6}$/')
    ->rule('email', 'email')
    ->check($data);
```

支持数组方式

```
Validate::make()
    ->rule(['zip' => '/^\d{6}$/', 'email' => 'email'])
    ->check($data);
```

验证规则可以使用`think\validate\ValidateRule`对象定义

```
Validate::make()
    ->rule('zip', ValidateRule::regex('/^\d{6}$/'))
    ->rule('email', ValidateRule::must()->isEmail())
```

```
    ->check($data);
```

> ValidateRule 类可以静态化调用所有的内置验证方法（支持二个参数：验证规则和错误信息）

下面是指定错误信息及验证字段的名称示例：

```
Validate::make()
    ->rule('zip', ValidateRule::regex('/^\d{6}$/', '邮编错误'))
    ->rule('email', ValidateRule::must()->isEmail()->title('邮箱'))
    ->rule('name', ValidateRule::min(5, '最少5个字符')->max(20, '不得超过20
个字符'))
    ->check($data);
```

同样在数组方式中也支持：

```
Validate::make()
    ->rule([
        'zip'   => ValidateRule::regex('/^\d{6}$/'),
        'email' => ValidateRule::email(),
    ])
    ->check($data);
```

验证数据

下面是一个典型的验证数据的例子：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
    'email' => 'email',
];

$msg = [
    'name.require' => '名称必须',
    'name.max'     => '名称最多不能超过25个字符',
    'age.number'   => '年龄必须是数字',
    'age.between'  => '年龄只能在1-120之间',
    'email'        => '邮箱格式错误',
];

$data = [
    'name'  => 'thinkphp',
    'age'   => 10,
    'email' => 'thinkphp@qq.com',
];
```

```
$validate   = Validate::make($rule,$msg);
$result = $validate->check($data);

if(!$result) {
    dump($validate->getError());
}
```

如果需要批量验证，可以使用：

```
$result   = Validate::make($rule,$msg)
->batch()
->check($data);
```

> 批量验证如果验证不通过，返回的是一个错误信息的数组。

闭包验证

可以对某个字段使用闭包验证，例如：

```
$validate = Validate::make([
    'name'  => function($value,$rule) {
        return $rule==$value ? true : false;
    },
])->check($data);
```

自定义验证规则

独立验证的时候支持使用 `extend` 方法动态注册验证规则，例如：

```
$data   = ['name' => 1];
$result = Validate::make(['name' => 'checkName:1'])
->extend('checkName', function ($value, $rule) {
    return $rule == $value ? true : '名称错误';
})->check($data);
```

支持批量注册验证规则，例如：

```
$data   = ['name' => 1];
$result = Validate::make(['name' => 'checkName:1'])
->extend([
    'checkName'=> function ($value, $rule) {
    return $rule == $value ? true : '名称错误';
},
    'checkStatus'=> [$this,'checkStatus']
```

```
])->check($data);
```

设置字段信息

验证类的架构方法支持传入`field`参数批量设置字段的描述信息，例如：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
    'email' => 'email',
];

$field = [
    'name'  => '名称',
    'age'   => '年龄',
    'email' => '邮箱',
];

$data = [
    'name'  => 'thinkphp',
    'age'   => 10,
    'email' => 'thinkphp@qq.com',
];

$validate = Validate::make($rule, [], $field);
$result   = $validate->check($data);
```

错误提示信息

如果要使用自定义的错误信息，可以使用`message`方法：

```
$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',
];

$validate = Validate::make($rule)->message($msg);
```

```
$result = $validate->check($data);

if(!$result){
    echo $validate->getError();
}
```

会输出 `年龄必须在1~120之间` 。

如果需要使用多语言验证信息，可以在定义验证信息的时候使用 {%语言变量} 替代 原来的验证错误信息，例如：

```
$rule = [
    'name'  => 'require|max:25',
    'age'   => 'number|between:1,120',
    'email' => 'email',
];

$msg = [
    'name.require' => '{%name_require}',
    'name.max'     => '{%name_max}',
    'age.number'   => '{%age_number}',
    'age.between'  => '{%age_between}',
    'email'        => '{%email_error}',
];

$data = [
    'name'  => 'thinkphp',
    'age'   => 121,
    'email' => 'thinkphp@qq.com',
];

$validate = Validate::make($rule)->message($msg);
$result = $validate->check($data);

if(!$result){
    echo $validate->getError();
}
```

静态调用

如果需要使用内置的规则验证单个数据，可以使用静态调用的方式。

> 要支持静态调用的话，必须使用 think\facade\Validate 类。

```
// 日期格式验证
Validate::dateFormat('2016-03-09','Y-m-d'); // true
// 验证是否有效的日期
Validate::isDate('2016-06-03'); // true
// 验证是否有效邮箱地址
Validate::isEmail('thinkphp@qq.com'); // true
// 验证是否在某个范围
Validate::in('a',['a','b','c']); // true
// 验证是否大于某个值
Validate::gt(10,8); // true
// 正则验证
Validate::regex(100,'\d+'); // true
```

> 静态验证的返回值为布尔值，错误信息需要自己判断返回值后处理。

更多验证规则可以参考前面的内置规则。

如果需要批量验证规则，可以使用

```
Validate::checkRule($value,'must|email');
```

> `checkRule` 方法始终返回布尔值，而不支持获取错误信息。

并且支持使用规则定义（需要引入 `think\validate\ValidateRule` 类）：

```
Validate::checkRule($value, ValidateRule::must()->isEmail());
```

表单令牌

验证规则支持对表单的令牌验证，首先需要在你的表单里面增加下面隐藏域：

```
<input type="hidden" name="__token__" value="{$Request.token}" />
```

或者

```
{:token()}
```

然后在你的验证规则中，添加`token`验证规则即可，例如，如果使用的是验证器的话，可以改为：

```
    protected $rule = [
        'name'  =>  'require|max:25|token',
        'email' =>  'email',
    ];
```

如果你的令牌名称不是`__token__`，则表单需要改为：

```
<input type="hidden" name="__hash__" value="{$Request.token.__hash__}" />
```

或者：

```
{:token('__hash__')}
```

验证器中需要改为：

```
    protected $rule = [
        'name'  =>  'require|max:25|token:__hash__',
        'email' =>  'email',
    ];
```

如果需要自定义令牌生成规则，可以调用`Request`类的`token`方法，例如：

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

```
use think\Controller;

class Index extends Controller
{
    public function index()
    {
        $token = $this->request->token('__token__', 'sha1');
        $this->assign('token', $token);
        return $this->fetch();
    }
}
```

然后在模板表单中使用：

```
<input type="hidden" name="__token__" value="{$token}" />
```

或者不需要在控制器写任何代码，直接在模板中使用：

```
{:token('__token__', 'sha1')}
```

杂项

缓存

概述

ThinkPHP采用`think\Cache`类（实际使用`think\facade\Cache`类即可）提供缓存功能支持。

内置支持的缓存类型包括file、memcache、wincache、sqlite、redis和xcache。

设置

全局的缓存配置直接修改配置目录下面的`cache.php`文件。

公共的缓存配置参数包含：

参数名	描述
type	缓存类型或者缓存驱动类名
expire	缓存有效期（秒）
prefix	缓存标识前缀
serialize	（非标量）是否需要自动序列化

不同的缓存驱动还需要配置额外的缓存参数。

如果你的应用只需要使用一种缓存类型，那么可以直接配置。

```
return [
    // 缓存类型为File
    'type'   =>  'file',
    // 全局缓存有效期（0为永久有效）
    'expire'=>  0,
    // 缓存前缀
    'prefix'=>  'think',
     // 缓存目录
    'path'   =>  '../runtime/cache/',
];
```

如果你需要同时使用多个缓存类型，那么可以配置为：

```
return [
    // 缓存配置为复合类型
    'type'   =>  'complex',
    'default'   =>      [
      'type'    =>      'file',
```

```
        // 全局缓存有效期（0为永久有效）
        'expire'=>  0,
        // 缓存前缀
        'prefix'=>  'think',
         // 缓存目录
        'path'  =>  '../runtime/cache/',
    ],
    'redis'     =>      [
        'type'    =>      'redis',
        'host'    =>      '127.0.0.1',
        // 全局缓存有效期（0为永久有效）
        'expire'=>  0,
        // 缓存前缀
        'prefix'=>  'think',
    ],
    // 添加更多的缓存类型设置
];
```

没有指定缓存类型的话，默认读取的是`default`缓存配置

```
// 使用文件缓存
Cache::set('name','value',3600);
Cache::get('name');

// 使用Redis缓存
Cache::store('redis')->set('name','value',3600);
Cache::get('name');

// 切换到文件缓存
Cache::store('default')->set('name','value',3600);
Cache::get('name');
```

还有一种方式是调用`connect`方法动态切换缓存。

```
$options = [
    // 缓存类型为File
    'type'  =>  'File',
    // 缓存有效期为永久有效
    'expire'=>  0,
    //缓存前缀
    'prefix'=>  'think',
     // 指定缓存目录
    'path'  =>  '../runtime/cache/',
];
Cache::connect($options)->set('name','value',3600);
Cache::connect($options)->get('name');
```

缓存参数根据不同的缓存方式会有所区别，通用的缓存参数如下：

参数	描述
type	缓存类型
expire	缓存有效期 （默认为0 表示永久缓存）
prefix	缓存前缀（默认为空）

使用

设置缓存

设置缓存有效期

```
Cache::set('name',$value,3600);
```

如果设置成功返回true，否则返回false。

缓存自增

针对数值类型的缓存数据，可以使用自增操作，例如：

```
// name自增（步进值为1）
Cache::inc('name');
// name自增（步进值为3）
Cache::inc('name',3);
```

缓存自减

针对数值类型的缓存数据，可以使用自减操作，例如：

```
// name自减（步进值为1）
Cache::dec('name');
// name自减（步进值为3）
Cache::dec('name',3);
```

获取缓存

获取缓存数据可以使用：

```
dump(Cache::get('name'));
```

如果`name`值不存在，则默认返回`false`。

支持指定默认值，例如：

```
dump(Cache::get('name',''));
```

表示如果`name`值不存在，则返回空字符串。

删除缓存

```
Cache::rm('name');
```

获取并删除缓存

```
Cache::pull('name');
```

如果`name`值不存在，则返回`null`。

清空缓存

```
Cache::clear();
```

不存在则写入缓存数据后返回

```
Cache::remember('name',function(){
        return time();
});
```

获取缓存对象

可以获取缓存对象，并且调用驱动类的高级方法，例如：

```
$cache = Cache::init();
// 获取缓存对象句柄
$handler = $cache->handler();
```

助手函数

系统对缓存操作提供了助手函数`cache`，用法如下：

```
$options = [
     // 缓存类型为File
    'type'   => 'File',
     // 缓存有效期为永久有效
    'expire' => 0,
     // 指定缓存目录
    'path'   => APP_PATH . 'runtime/cache/',
];

// 缓存初始化
// 不进行缓存初始化的话，默认使用配置文件中的缓存配置
```

```
cache($options);

// 设置缓存数据
cache('name', $value, 3600);
// 获取缓存数据
var_dump(cache('name'));
// 删除缓存数据
cache('name', NULL);
```

缓存标签

支持给缓存数据打标签，例如：

```
Cache::tag('tag')->set('name1','value1');
Cache::tag('tag')->set('name2','value2');

// 或者批量设置缓存标签
Cache::set('name1','value1');
Cache::set('name2','value2');
Cache::tag('tag',['name1','name2']);

// 清除tag标签的缓存数据
Cache::clear('tag');
```

同时使用多个缓存类型

如果要同时使用多个缓存类型进行操作的话，可以做如下配置：

```
return [
    // 使用复合缓存类型
    'type'  =>  'complex',
    // 默认使用的缓存
    'default'   =>  [
        // 驱动方式
        'type'   => 'file',
        // 缓存保存目录
        'path'   => '../runtime/default',
    ],
    // 文件缓存
    'file'   =>  [
        // 驱动方式
        'type'   => 'file',
        // 设置不同的缓存保存目录
        'path'   => '../runtime/file/',
    ],
    // redis缓存
    'redis'   =>  [
        // 驱动方式
        'type'   => 'redis',
        // 服务器地址
        'host'       => '127.0.0.1',
    ],
```

```
    ],
```

`type`配置为`complex`之后，就可以缓存多个缓存类型和缓存配置，每个缓存配置的方法和之前一样，并且你可以给相同类型的缓存类型（使用不同的缓存标识）配置不同的缓存配置参数。

当使用

```
Cache::set('name', 'value');
Cache::get('name');
```

的时候，其实使用的是`default`缓存标识的缓存配置，如果需要切换到其它的缓存标识操作，可以使用：

```
// 切换到file操作
Cache::store('file')->set('name','value');
Cache::get('name');
// 切换到redis操作
Cache::store('redis')->set('name','value');
Cache::get('name');
```

如果要返回当前缓存类型对象的句柄，可以使用

```
// 获取Redis对象 进行额外方法调用
Cache::store('redis')->handler();
```

Session

概述

可以直接使用`think\facade\Session`类操作`Session`。

版本	新增功能
5.1.3	`get`方法支持获取多级

Session初始化

Session会在第一次调用Session类的时候按照`session.php`配置的参数自动初始化：

```
return [
    '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');
```

或者调用init方法进行初始化：

```
Session::init([
    'prefix'         => 'module',
    'type'           => '',
    'auto_start'     => true,
]);
```

> 如果你没有使用Session类进行Session操作的话，例如直接操作`$_SESSION`,必须使用上面的方式手动初始化或者直接调用`session_start()`方法进行`session`初始化。

设置参数

默认支持的session设置参数包括：

参数	描述
type	session类型
expire	session过期时间
prefix	session前缀
auto_start	是否自动开启
use_trans_sid	是否使用use_trans_sid
var_session_id	请求session_id变量名
id	session_id
name	session_name
path	session保存路径
domain	session cookie_domain
use_cookies	是否使用cookie
cache_limiter	session_cache_limiter
cache_expire	session_cache_expire
secure	安全选项
httponly	使用httponly

> 如果做了session驱动扩展，可能有些参数不一定有效。

基础用法

赋值

```
// 赋值（当前作用域）
Session::set('name','thinkphp');
// 赋值think作用域
Session::set('name','thinkphp','think');
```

判断是否存在

```
// 判断（当前作用域）是否赋值
Session::has('name');
// 判断think作用域下面是否赋值
Session::has('name','think');
```

取值

```
// 取值（当前作用域）
```

```
Session::get('name');
// 取值think作用域
Session::get('name','think');
```

如果name的值不存在，返回 `null` 。

删除

```
// 删除（当前作用域）
Session::delete('name');
// 删除think作用域下面的值
Session::delete('name','think');
```

指定作用域

```
// 指定当前作用域
Session::prefix('think');
```

取值并删除

```
// 取值并删除
Session::pull('name');
```

如果name的值不存在，返回 `Null` 。

清空

```
// 清除session（当前作用域）
Session::clear();
// 清除think作用域
Session::clear('think');
```

闪存数据，下次请求之前有效

```
// 设置session 并且在下一次请求之前有效
Session::flash('name','value');
```

提前清除当前请求有效的数据

```
// 清除当前请求有效的session
Session::flush();
```

二级数组

支持session的二维数组操作，例如：

```
// 赋值（当前作用域）
Session::set('name.item','thinkphp');
// 判断（当前作用域）是否赋值
Session::has('name.item');
// 取值（当前作用域）
Session::get('name.item');
// 删除（当前作用域）
Session::delete('name.item');
```

助手函数

系统也提供了助手函数session完成相同的功能，例如：

```
// 初始化session
session([
    'prefix'     => 'module',
    'type'       => '',
    'auto_start' => true,
]);

// 赋值（当前作用域）
session('name', 'thinkphp');

// 赋值think作用域
session('name', 'thinkphp', 'think');

// 判断（当前作用域）是否赋值
session('?name');

// 取值（当前作用域）
session('name');

// 取值think作用域
session('name', '', 'think');

// 删除（当前作用域）
session('name', null);

// 清除session（当前作用域）
session(null);

// 清除think作用域
session(null, 'think');
```

Session驱动

支持指定 Session 驱动，配置文件如下：

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

表示使用`redis`作为`session`类型。

目前内置支持使用`redis`、`memcache`和`memcached`作为session驱动类型。

Cookie

概述

ThinkPHP采用`think\facade\Cookie`类提供Cookie支持。

配置

配置文件位于配置目录下的`cookie.php`文件，无需手动初始化，系统会自动在调用之前进行Cookie初始化工作。

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]);
```

永久保存

```
// 永久保存Cookie
Cookie::forever('name','value');
```

判断

```
Cookie::has('name');
// 判断指定前缀的cookie值是否存在
Cookie::has('name','think_');
```

获取

```
Cookie::get('name');
// 获取指定前缀的cookie值
Cookie::get('name','think_');
```

删除

```
//删除cookie
Cookie::delete('name');
// 删除指定前缀的cookie
Cookie::delete('name','think_');
```

清空

```
// 清空指定前缀的cookie
Cookie::clear('think_');
```

> 如果不指定前缀，不能做清空操作

助手函数

系统提供了cookie助手函数用于基本的cookie操作，例如：

```
// 初始化
cookie(['prefix' => 'think_', 'expire' => 3600]);

// 设置
cookie('name', 'value', 3600);

// 获取
echo cookie('name');

// 删除
cookie('name', null);
```

```
// 清除
cookie(null, 'think_');
```

多语言

ThinkPHP内置通过 `\think\facade\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

默认情况下，系统载入的是配置的默认语言包，并且不会自动侦测当前系统的语言。

默认语言由 `default_lang` 配置参数设置，系统默认设置为：

```
// 默认语言
'default_lang'           => 'zh-cn',
```

要启用语言自动侦测和多语言自动切换，需要开启多语言切换，在应用的公共配置文件添加：

```
// 开启语言切换
'lang_switch_on' => true,
```

开启后，系统会自动检测当前语言（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允许的语言列表内认为有效，否则使用默认设置的语言

如果不希望浏览器自动侦测语言，请关闭 `lang_switch_on` 后设置默认语言。

如果在自动侦测语言的时候，希望设置允许的语言列表，不在列表范围的语言则仍然使用默认语言，可以使用：

```
// 设置允许的语言
Lang::setAllowLangList(['zh-cn','en-us']);
```

语言变量定义

语言变量的定义，只需要在需要使用多语言的地方，写成：

```
Lang::get('add user error');
// 使用系统封装的助手函数
lang('add user error');
```

也就是说，字符串信息要改成`Lang::get`方法来表示。

语言定义一般采用英语来描述。

语言文件定义

系统会默认加载下面三个语言包：

```
框架语言包: thinkphp\lang\当前语言.php
应用语言包: application\lang\当前语言.php
模块语言包: application\模块\lang\当前语言.php
```

如果你还需要加载其他的语言包，可以在设置或者自动检测语言之后，用load方法进行加载：

```
Lang::load( '../application/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',
];
```

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。

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！'），就可以使用下面的字符串来替代原来的错误提示。

```
{%lang_var}
```

如果要在模板中输出语言变量不需要在控制器中赋值，可以直接使用模板引擎特殊标签来直接输出语言定义的值：

```
{$Think.lang.lang_var}
```

可以输出当前语言包里面定义的 `lang_var` 语言定义。

变量传入支持

语言包定义的时候支持传入变量，有两种方式

使用命名绑定方式，例如：

```
'file_format'    =>    '文件格式: {:format},文件大小：{:size}',
```

在模板中输出语言字符串的时候传入变量值即可：

```
{:lang('file_format',['format' => 'jpeg,png,gif,jpg','size' => '2MB'])}
```

第二种方式是使用格式字串，如果你需要使用第三方的翻译工具，建议使用该方式定义变量。

```
'file_format'    =>    '文件格式: %s,文件大小：%d',
```

在模板中输出多语言的方式更改为：

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

分页

分页实现

`ThinkPHP5.1`内置了分页实现，要给数据添加分页输出功能变得非常简单，可以直接在`Db`类查询的时候调用`paginate`方法：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = Db::name('user')->where('status',1)->paginate(10);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

也可以改成模型的分页查询代码：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

模板文件中分页输出代码如下：

```
<div>
<ul>
{volist name='list' id='user'}
    <li> {$user.nickname}</li>
{/volist}
</ul>
</div>
{$list|raw}
```

也可以单独赋值分页输出的模板变量

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10);
// 获取分页显示
$page = $list->render();
// 模板变量赋值
$this->assign('list', $list);
$this->assign('page', $page);
// 渲染模板输出
```

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

模板文件中分页输出代码如下：

```
<div>
<ul>
{volist name='list' id='user'}
    <li> {$user.nickname}</li>
{/volist}
</ul>
</div>
{$page|raw}
```

默认情况下，生成的分页输出是完整分页功能，带总分页数据和上下页码，分页样式只需要通过样式修改即可，完整分页默认生成的分页输出代码为：

```
<ul class="pagination">
<li><a href="?page=1">&laquo;</a></li>
<li><a href="?page=1">1</a></li>
<li class="active"><span>2</span></li>
<li class="disabled"><span>&raquo;</span></li>
</ul>
```

如果你需要单独获取总的数据，可以使用

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10);
// 获取总记录数
$count = $list->total();
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

传入总记录数

支持传入总记录数而不会自动进行总数计算，例如：

```
// 查询状态为1的用户数据 并且每页显示10条数据 总记录数为1000
$list = User::where('status',1)->paginate(10,1000);
// 获取分页显示
$page = $list->render();
// 模板变量赋值
$this->assign('list', $list);
$this->assign('page', $page);
// 渲染模板输出
```

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

> 对于 UNION 查询以及一些特殊的复杂查询，推荐使用这种方式首先单独查询总记录数，然后再传入分页方法

分页后数据处理

支持分页类后数据直接 each 遍历处理，方便修改分页后的数据，而不是只能通过模型的获取器来补充字段。

```
$list = User::where('status',1)->paginate()->each(function($item, $key){
    $item->nickname = 'think';
});
```

如果是 Db 类操作分页数据的话， each 方法的闭包函数中需要使用返回值，例如：

```
$list = Db::name('user')->where('status',1)->paginate()->each(function($i
tem, $key){
    $item['nickname'] = 'think';
    return $item;
});
```

简洁分页

如果你仅仅需要输出一个 仅仅只有上下页的分页输出，可以使用下面的简洁分页代码：

```
// 查询状态为1的用户数据 并且每页显示10条数据
$list = User::where('status',1)->paginate(10,true);
// 把分页数据赋值给模板变量list
$this->assign('list', $list);
// 渲染模板输出
return $this->fetch();
```

简洁分页模式的输出代码为：

```
<ul class="pager">
<li><a href="?page=1">&laquo;</a></li>
<li class="disabled"><span>&raquo;</span></li>
</ul>
```

> 由于简洁分页模式不需要查询总数据数，因此可以提高查询性能。

分页参数

主要的分页参数如下：

参数	描述
list_rows	每页数量
page	当前页
path	url路径
query	url额外参数
fragment	url锚点
var_page	分页变量
type	分页类名

分页参数的设置方式有两种，第一种是在配置文件`paginate.php`中定义，例如：

```
//分页配置
return [
    'type'     => 'bootstrap',
    'var_page' => 'page',
];
```

type属性支持命名空间，例如：

```
//分页配置
return [
    'type'     => '\org\page\bootstrap',
    'var_page' => 'page',
];
```

也可以在调用分页方法的时候传入，例如：

```
$list = Db::name('user')->where('status',1)->paginate(10,true,[
    'type'     => 'bootstrap',
    'var_page' => 'page',
]);
```

> 如果需要在分页的时候传入查询条件，可以使用query参数拼接额外的查询参数

上传

上传文件

> 内置的上传只是上传到本地服务器，上传到远程或者第三方平台的话需要自己扩展。

假设表单代码如下：

```
<form action="/index/index/upload" enctype="multipart/form-data" method="
post">
<input type="file" name="image" /> <br>
<input type="submit" value="上传" />
</form>
```

然后在控制器中添加如下的代码：

```
public function upload(){
    // 获取表单上传文件 例如上传了001.jpg
    $file = request()->file('image');
    // 移动到框架应用根目录/uploads/ 目录下
    $info = $file->move( '../uploads');
    if($info){
        // 成功上传后 获取上传信息
        // 输出 jpg
        echo $info->getExtension();
        // 输出 20160820/42a79759f284b767dfcb2a0197904287.jpg
        echo $info->getSaveName();
        // 输出 42a79759f284b767dfcb2a0197904287.jpg
        echo $info->getFilename();
    }else{
        // 上传失败获取错误信息
        echo $file->getError();
    }
}
```

`move`方法成功的话返回的是一个`\think\File`对象，你可以对上传后的文件进行后续操作。

多文件上传

如果你使用的是多文件上传表单，例如：

```
<form action="/index/index/upload" enctype="multipart/form-data" method="
post">
```

```
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<input type="file" name="image[]" /> <br>
<input type="submit" value="上传" />
</form>
```

控制器代码可以改成：

```
public function upload(){
    // 获取表单上传文件
    $files = request()->file('image');
    foreach($files as $file){
        // 移动到框架应用根目录/uploads/ 目录下
        $info = $file->move( '../uploads');
        if($info){
            // 成功上传后 获取上传信息
            // 输出 jpg
            echo $info->getExtension();
            // 输出 42a79759f284b767dfcb2a0197904287.jpg
            echo $info->getFilename();
        }else{
            // 上传失败获取错误信息
            echo $file->getError();
        }
    }
}
```

上传验证

支持对上传文件的验证，包括文件大小、文件类型和后缀：

```
public function upload(){
    // 获取表单上传文件 例如上传了001.jpg
    $file = request()->file('image');
    // 移动到框架应用根目录/uploads/ 目录下
    $info = $file->validate(['size'=>15678,'ext'=>'jpg,png,gif'])->move(
'../uploads');
    if($info){
        // 成功上传后 获取上传信息
        // 输出 jpg
        echo $info->getExtension();
        // 输出 20160820/42a79759f284b767dfcb2a0197904287.jpg
        echo $info->getSaveName();
        // 输出 42a79759f284b767dfcb2a0197904287.jpg
        echo $info->getFilename();
    }else{
        // 上传失败获取错误信息
        echo $file->getError();
    }
}
```

> `getSaveName` 方法返回的是图片的服务器文件地址，并不能直接用于图片的URL地址，尤其在windows平台上必须做转换才能正常显示图片。

如果上传文件验证不通过，则 `move` 方法返回false。

验证参数	说明
size	上传文件的最大字节
ext	文件后缀，多个用逗号分割或者数组
type	文件MIME类型，多个用逗号分割或者数组

> 还有一个额外的自动验证规则是，如果上传的文件后缀是图像文件后缀，则会检查该文件是否是一个合法的图像文件。

> 上传错误提示信息支持多语言，你可以修改语言包来修改错误提示。

上传规则

默认情况下，会在上传目录下面生成以当前日期为子目录，以微秒时间的 `md5` 编码为文件名的文件，例如上面生成的文件名可能是：

```
/home/www/uploads/20160510/42a79759f284b767dfcb2a0197904287.jpg
```

我们可以指定上传文件的命名规则，使用 `rule` 方法即可，例如：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用md5规则
$file->rule('md5')->move('../uploads/');
```

最终生成的文件名类似于：

```
application/uploads/72/ef580909368d824e899f77c7c98388.jpg
```

系统默认提供了几种上传命名规则，包括：

规则	描述
date	根据日期和微秒数生成

md5	对文件使用md5_file散列生成
sha1	对文件使用sha1_file散列生成

> 其中md5和sha1规则会自动以散列值的前两个字符作为子目录，后面的散列值作为文件名。

如果需要使用自定义命名规则，可以在`rule`方法中传入函数或者方法，例如：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用uniqid规则
$file->rule('uniqid')->move('../uploads/');
```

生成的文件名类似于：

```
application/uploads/573d3b6d7abe2.jpg
```

如果你希望保留原文件名称，可以使用：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用原文件名
$file->move('../uploads/','');
```

默认情况下，会覆盖服务器上传目录下的同名文件，如果不希望覆盖，可以使用：

```
// 获取表单上传文件 例如上传了001.jpg
$file = request()->file('image');
// 移动到服务器的上传目录 并且设置不覆盖
$file->move('../uploads/',true,false);
```

获取文件hash散列值

可以获取上传文件的哈希散列值，例如：

```
// 获取表单上传文件
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用原文件名
$upload = $file->move('/home/www/upload/');
// 获取上传文件的hash散列值
echo $upload->md5();
```

```
echo $upload->sha1();
```

可以统一使用hash方法获取文件散列值

```
// 获取表单上传文件
$file = request()->file('image');
// 移动到服务器的上传目录 并且使用原文件名
$upload = $file->move('/home/www/upload/');
// 获取上传文件的hash散列值
echo $upload->hash('sha1');
echo $upload->hash('md5');
```

返回对象

上传成功后返回的仍然是一个`File`对象，除了File对象自身的方法外，并且可以使用`SplFileObject`的属性和方法，便于进行后续的文件处理。

命令行

ThinkPHP5.1支持`Console`应用，通过命令行的方式执行一些URL访问不方便或者安全性较高的操作。

我们可以在cmd命令行下面，切换到应用根目录（注意不是web根目录），然后执行`php think`，会出现下面的提示信息：

```
>php think
Think Console version 0.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -V, --version         Display this console version
  -q, --quiet           Do not output any message
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal
output, 2 for more verbose output and 3 for debug

Available commands:
  build              Build Application Dirs
  clear              Clear runtime file
  help               Displays help for a command
  list               Lists commands
 make
  make:controller    Create a new resource controller class
  make:model         Create a new model class
 optimize
  optimize:autoload  Optimizes PSR0 and PSR4 packages to be loaded with c
lassmaps too, good for production.
  optimize:config    Build config and common file cache.
  optimize:route     Build route cache.
```

`console`命令的执行格式一般为：

> >php think 指令 参数

下面介绍下系统自带的几个命令，包括：

指令	描述

build	自动生成目录和文件
help	帮助
list	指令列表
clear	清除缓存指令
make:controller	创建控制器文件
make:model	创建模型文件
optimize:autoload	生成类库映射文件
optimize:config	生成配置缓存文件
optimize:schema	生成数据表字段缓存文件

更多的指令可以自己扩展。

启动内置服务器

启动内置服务器

`V5.1.5+`版本开始，增加了启动内置服务器的指令，方便测试。

命令行切换到应用根目录后，输入：

```
>php think run
```

如果启动成功，会输出下面信息，并显示`web`目录位置。

```
ThinkPHP Development server is started On <http://127.0.0.1:8000/>
You can exit with `CTRL-C`
Document root is: D:\WWW\tp5/public
```

然后你可以直接在浏览器里面访问

```
http://127.0.0.1:8000/
```

而无须设置`Vhost`，不过需要注意，这个只有web服务器，其它的例如数据库服务的需要自己单独管理。

自动生成目录结构

ThinkPHP5.1 具备自动创建功能，可以用来自动生成需要的模块及目录结构和文件等。

快速生成模块

生成一个 `test` 模块的指令如下：

```
>php think build --module test
```

表示自动生成 `test` 模块，自动生成的模块目录包含了 `config` 、 `controller` 、 `model` 和 `view` 目录以及 `common.php` 公共文件。

批量生成模块

如果需要批量生成多个模块的目录和文件，需要定义规则文件 `build.php` 并放入应用目录下面。

默认的框架的根目录下面自带了一个 `build.php` 示例参考文件（把该文件修改后放入应用根目录下面即可），内容如下：

```
return [
    // 生成应用公共文件
    '__file__' => ['common.php'],

    // 定义demo模块的自动生成 （按照实际定义的文件名生成）
    'demo'     => [
        '__file__'   => ['common.php'],
        '__dir__'    => ['behavior', 'controller', 'model', 'view'],
        'controller' => ['Index', 'Test', 'UserType'],
        'model'      => ['User', 'UserType'],
        '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）

自动生成以应用目录为起始目录，`__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 应用目录/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。

我们还可以在应用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 [
    // 定义demo模块的自动生成
    'demo'   => [
        '__file__'   => ['tags.php', 'user.php', 'hello.php'],
        '__dir__'    => ['config', 'controller', 'model', 'view'],
        'controller' => ['Index', 'Test', 'UserType'],
        'model'      => [],
        'view'       => ['index/index'],
    ],

    // 定义test模块的自动生成
    'test'=>[
        '__dir__'   =>  ['config','controller','model','widget'],
        'controller'=>  ['Index','Test','UserType'],
        'model'     =>   ['User','UserType'],
        'view'      =>  ['index/index','index/test'],
    ],
 ];
```

定义好生成规则文件后，我们在命令行下面输入命令：

```
>php think build
```

如果看到输出

```
Successed
```

则表示自动生成成功。

创建类库文件

快速生成控制器

执行下面的指令可以生成`index`模块的`Blog`控制器类库文件

```
>php think make:controller index/Blog
```

默认生成的是一个资源控制器，类文件如下：

```
<?php

namespace app\index\controller;

use think\Controller;
use think\Request;

class Blog extends Controller
{
    /**
     * 显示资源列表
     *
     * @return \think\Response
     */
    public function index()
    {
        //
    }

    /**
     * 显示创建资源表单页.
     *
     * @return \think\Response
     */
    public function create()
    {
        //
    }

    /**
     * 保存新建的资源
     *
     * @param  \think\Request  $request
     * @return \think\Response
     */
    public function save(Request $request)
    {
        //
    }
```

```
    /**
     * 显示指定的资源
     *
     * @param  int  $id
     * @return \think\Response
     */
    public function read($id)
    {
        //
    }

    /**
     * 显示编辑资源表单页.
     *
     * @param  int  $id
     * @return \think\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * 保存更新的资源
     *
     * @param  \think\Request  $request
     * @param  int  $id
     * @return \think\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * 删除指定资源
     *
     * @param  int  $id
     * @return \think\Response
     */
    public function delete($id)
    {
        //
    }
}
```

默认生成的控制器类继承`\think\Controller`，并且生成了资源操作方法，如果仅仅生成空的控制器则可以使用：

```
>php think make:controller index/Blog --plain
```

生成的控制器类文件如下：

```
<?php

namespace app\index\controller;

use think\Controller;

class Blog extends Controller
{
    //
}
```

如果需要生成多级控制器，可以使用

```
>php think make:controller index/test/Blog --plain
```

会生成一个 `app\index\test\Blog` 控制器类。

快速生成模型

和生成控制器类似，执行下面的指令可以生成 `index` 模块的 `Blog` 模型类库文件

```
>php think make:model index/Blog
```

生成的模型类文件如下：

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

use think\Model;

class Blog extends Model
{
    //
}
```

生成带后缀的类库

如果要生成带后缀的类库，可以直接使用：

```
>php think make:controller index/BlogController
```

```
>php think make:model index/BlogModel
```

生成类库映射文件

生成类库映射文件`optimize:autoload`

可以使用下面的指令生成类库映射文件，提高系统自动加载的性能。

```
>php think optimize:autoload
```

指令执行成功后，会在rumtime目录下面生成`classmap.php`文件，生成的类库映射文件会扫描系统目录和应用目录的类库。

清除缓存文件

清除缓存文件`clear`

如果需要清除应用的缓存文件，可以使用下面的命令：

```
php think clear
```

不带任何参数调用clear命令的话，会清除`runtime`目录（包括模板缓存、日志文件及其子目录）下面的所有的文件，但会保留目录。

如果需要清除某个指定目录下面的文件，可以使用：

```
php think clear --path d:\www\tp5\runtime\log\
```

生成配置缓存文件

生成配置缓存 `optimize:config`

可以为应用或者模块生成配置缓存文件

```
php think optimize:config
```

默认生成应用的配置缓存文件，调用后会在 `runtime` 目录下面生成 `init.php` 文件，生成配置缓存文件后，应用目录下面的 `config.php` `common.php` 以及 `tags.php` 不会被加载，被 `runtime/init.php` 取代。

如果需要生成某个模块的配置缓存，可以使用：

```
php think optimize:config index
```

调用后会在 `runtime/index` 目录下面生成 `init.php` 文件，生成后， `index` 模块目录下面的 `config.php` `common.php` 以及 `tags.php` 不会被加载，被 `runtime/index/init.php` 取代。

生成数据表字段缓存

生成数据表字段缓存 `optimize:schema`

可以通过生成数据表字段信息缓存，提升数据库查询的性能，避免多余的查询。命令如下：

```
php think optimize:schema
```

会自动生成当前数据库配置文件中定义的数据表字段缓存，也可以指定数据库生成字段缓存（必须有用户权限），例如，下面指定生成 `demo` 数据库下面的所有数据表的字段缓存信息。

```
php think optimize:schema --db demo
```

执行后会自动在 `runtime/schema` 目录下面按照数据表生成字段缓存文件。

如果你的应用使用了不同的数据库连接，可以根据模块来生成，如下：

```
php think optimize:schema --module index
```

会读取index模块的模型来生成数据表字段缓存。

> 没有继承think\Model类的（抽象）模型类不会生成。

更新数据表字段缓存也是同样的方式，每次执行都会重新生成缓存。如果需要单独更新某个数据表的缓存，可以使用：

```
php think optimize:schema --table think_user
```

支持指定数据库名称

```
php think optimize:schema --table demo.think_user
```

生成路由映射缓存

生成路由映射缓存 `optimize:route`

> 路由映射缓存用于开启路由延迟解析的情况下，支持路由反解的URL生成，如果你没有开启路由延迟解析或者没有使用URL路由反解生成则不需要生成。

生成路由映射缓存的命令：

```
php think optimize:route
```

执行后，会在 `runtime` 目录下面生成 `route.php` 文件。

自定义指令

创建自定义指令

第一步，创建一个自定义命令类文件，新建 `application/common/command/Hello.php`

```
<?php
namespace app\common\command;

use think\console\Command;
use think\console\Input;
use think\console\input\Argument;
use think\console\input\Option;
use think\console\Output;

class Hello extends Command
{
    protected function configure()
    {
        $this->setName('hello')
                ->addArgument('name', Argument::OPTIONAL, "your name")
            ->addOption('city', null, Option::VALUE_REQUIRED, 'city name'
)
                ->setDescription('Say Hello');
    }

    protected function execute(Input $input, Output $output)
    {
        $name = trim($input->getArgument('name'));
        $name = $name ?: 'thinkphp';

                if ($input->hasOption('city')) {
                $city = PHP_EOL . 'From ' . $input->getOption('city');
        } else {
                $city = '';
        }

        $output->writeln("Hello," . $name . '!' . $city);
    }
}
```

这个文件定义了一个叫 `hello` 的命令，并设置了一个 `name` 参数和一个 `city` 选项。

第二步，配置 `application/command.php` 文件

```
<?php
return [
```

```
    'app\common\command\Hello',
];
```

第三步，测试-命令帮助-命令行下运行

```
php think
```

输出

```
Think Console version 0.1

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  -V, --version         Display this console version
  -q, --quiet           Do not output any message
      --ansi            Force ANSI output
      --no-ansi         Disable ANSI output
  -n, --no-interaction  Do not ask any interactive question
  -v|vv|vvv, --verbose  Increase the verbosity of messages: 1 for normal
output, 2 for more verbose output and 3 for debug

Available commands:
  build              Build Application Dirs
  clear              Clear runtime file
  hello              Say Hello
  help               Displays help for a command
  list               Lists commands
 make
  make:controller    Create a new resource controller class
  make:model         Create a new model class
 optimize
  optimize:autoload  Optimizes PSR0 and PSR4 packages to be loaded with c
lassmaps too, good for production.
  optimize:config    Build config and common file cache.
  optimize:schema    Build database schema cache.
```

第四步，运行`hello`命令

```
php think hello
```

输出

```
Hello thinkphp!
```

添加命令参数

```
php think hello liuchen
```

输出

```
Hello liuchen!
```

添加`city`选项

```
php think hello liuchen --city shanghai
```

输出

```
Hello thinkphp!
From shanghai
```

> 注意看参数和选项的调用区别

扩展库

下面的扩展非核心内置，需要单独通过`composer`安装。

验证码

首先使用`Composer`安装`think-captcha`扩展包：

```
composer require topthink/think-captcha
```

验证码的简单用法

扩展包内定义了一些常见用法方便使用，可以满足大部分常用场景，以下示例说明。

在模版内添加验证码的显示代码

```
<div>{:captcha_img()}</div>
```

或者

```
<div><img src="{:captcha_src()}" alt="captcha" /></div>
```

> 上面两种的最终效果是一样的，根据需要调用即可。

然后使用框架的内置验证功能（具体可以参考验证章节），添加`captcha`验证规则即可

```
$this->validate($data,[
    'captcha|验证码'=>'require|captcha'
]);
```

如果没有使用内置验证功能，则可以调研内置的函数手动验证

```
if(!captcha_check($captcha)){
 // 验证失败
};
```

验证码的自定义用法

如果需要自己独立生成验证码，可以调用`Captcha`类（`think\captcha\Captcha`）操作。

在控制器中使用下面的代码进行验证码生成：

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

use think\captcha\Captcha;

class Index
{
        public function verify()
    {
        $captcha = new Captcha();
        return $captcha->entry();
    }

}
```

然后访问下面的地址就可以显示验证码：

```
http://serverName/index/index/verify
```

输出效果如图

通常可以给验证码地址注册路由

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

在模板中就可以使用下面的代码显示验证码图片

```
<div><img src="{:url('index/verify')}" alt="captcha" /></div>
```

如果你需要在一个页面中生成多个验证码的话，`entry`方法需要传入可标识的信息（数字或者字符串），例如：

```
$captcha = new Captcha();
return $captcha->entry(1);
```

可以用 `Captcha` 类的 `check` 方法检测验证码的输入是否正确，

```
// 检测输入的验证码是否正确，$value为用户输入的验证码字符串
$captcha = new Captcha();
if( !$captcha->check($value))
{
        // 验证失败
}
```

或者直接调用封装的一个验证码检测的函数 `captcha_check`

```
// 检测输入的验证码是否正确，$value为用户输入的验证码字符串
if( !captcha_check($value ))
{
        // 验证失败
}
```

如果你在页面上同时生成了多个验证码，则可以使用

```
// 检测输入的验证码是否正确，$value为用户输入的验证码字符串，$id为验证码标识
if( !captcha_check($value, $id ))
{
        // 验证失败
}
```

验证码的配置参数

`Captcha` 类带有默认的配置参数，支持自定义配置。这些参数包括：

参数	描述	默认
codeSet	验证码字符集合	略
expire	验证码过期时间（s）	1800
useZh	使用中文验证码	false
zhSet	中文验证码字符串	略
useImgBg	使用背景图片	false
fontSize	验证码字体大小(px)	25
useCurve	是否画混淆曲线	true
useNoise	是否添加杂点	true
imageH	验证码图片高度，设置为0为自动计算	0
imageW	验证码图片宽度，设置为0为自动计算	0
length	验证码位数	5

fontttf	验证码字体，不设置是随机获取	空
bg	背景颜色	[243, 251, 254]
reset	验证成功后是否重置	true

> 如果使用扩展内置的方法进行验证码显示，直接在应用的 `config` 目录下面的 `captcha.php` 文件（没有则首先创建）中进行设置即可，以下设置方式仅限于独立调用Captcha类的时候使用。

实例化传入参数：

```
$config =    [
    // 验证码字体大小
    'fontSize'    =>    30,
    // 验证码位数
    'length'      =>    3,
    // 关闭验证码杂点
    'useNoise'    =>    false,
];
$captcha = new Captcha($config);
return $captcha->entry();
```

或者采用动态设置的方式，如：

```
$captcha = new Captcha();
$captcha->fontSize = 30;
$captcha->length   = 3;
$captcha->useNoise = false;
return $captcha->entry();
```

验证码字体

默认情况下，验证码的字体是随机使用扩展包内 `think-captcha/assets/ttfs` 目录下面的字体文件，我们可以指定验证码的字体，例如：

```
$captcha = new Captcha();
$captcha->fontttf = '5.ttf';
return $captcha->entry();
```

背景图片

支持验证码背景图片功能，可以如下设置：

```
$captcha = new Captcha();
```

```
// 开启验证码背景图片功能 随机使用扩展包内`think-captcha/assets/bgs`目录下面的图片
$captcha->useImgBg = true;
return $captcha->entry();
```

中文验证码

如果要使用中文验证码，可以设置：

```
$captcha = new Captcha();
// 使用中文验证码（字体使用扩展包内`think-captcha/assets/zhttfs`字体文件）
$captcha->useZh = true;
return $captcha->entry();
```

指定验证码字符

指定验证码的字符，可以设置：

```
$captcha = new Captcha();
// 设置验证码字符为纯数字
$captcha->codeSet = '0123456789';
return $captcha->entry();
```

如果是中文验证码，可以使用`zhSet`参数设置，例如：

```
$captcha = new Captcha();
$captcha->useZh = true;
// 设置验证码字符
$captcha->zhSet = '们以我到他会作时要动国产的一是工就年阶义发成部民可出能方进在了不和有大这';
return $captcha->entry();
```

> 默认的验证码字符已经剔除了易混淆的 `1l0o` 等字符

图像处理

安装扩展

使用`Composer`安装`ThinkPHP5`的图像处理类库：

```
composer require topthink/think-image
```

图像操作

下面来看下图像操作类的基础方法。

打开图像文件

假设当前入口文件目录下面有一个`image.png`文件，如图所示：

使用`open`方法打开图像文件进行相关操作：

```
$image = \think\Image::open('./image.png');
```

也可以从直接获取当前请求中的文件上传对象，例如：

```
$image = \think\Image::open(request()->file('image'));
```

获取图像信息

可以获取打开图片的信息，包括图像大小、类型等，例如：

```
$image = \think\Image::open('./image.png');
// 返回图片的宽度
$width = $image->width();
// 返回图片的高度
$height = $image->height();
// 返回图片的类型
$type = $image->type();
// 返回图片的mime类型
$mime = $image->mime();
// 返回图片的尺寸数组 0 图片宽度 1 图片高度
$size = $image->size();
```

裁剪图片

使用`crop`和`save`方法完成裁剪图片功能。

```
$image = \think\Image::open('./image.png');
//将图片裁剪为300x300并保存为crop.png
$image->crop(300, 300)->save('./crop.png');
```

生成的图片如图：

支持从某个坐标开始裁剪，例如下面从（100，30）开始裁剪，例如：

```
$image = \think\Image::open('./image.png');
```

```
//将图片裁剪为300x300并保存为crop.png
$image->crop(300, 300,100,30)->save('./crop.png');
```

生成的图片如图：

生成缩略图

使用`thumb`方法生成缩略图，例如：

```
$image = \think\Image::open('./image.png');
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png
$image->thumb(150, 150)->save('./thumb.png');
```

生成的缩略图如图所示：

我们看到实际生成的缩略图并不是150*150，因为默认采用原图等比例缩放的方式生成缩略图，最大宽度是150。

可以支持其他类型的缩略图生成，设置包括`\think\Image`的下列常量或者对应的数字：

```
//常量，标识缩略图等比例缩放类型
const THUMB_SCALING   = 1;
//常量，标识缩略图缩放后填充类型
const THUMB_FILLED    = 2;
//常量，标识缩略图居中裁剪类型
const THUMB_CENTER    = 3;
//常量，标识缩略图左上角裁剪类型
const THUMB_NORTHWEST = 4;
//常量，标识缩略图右下角裁剪类型
const THUMB_SOUTHEAST = 5;
//常量，标识缩略图固定尺寸缩放类型
```

```
const THUMB_FIXED      = 6;
```

比如我们居中裁剪：

```
$image = \think\Image::open('./image.png');
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png
$image->thumb(150,150,\think\Image::THUMB_CENTER)->save('./thumb.png');
```

后生成的缩略图效果如图：

再比如我们右下角剪裁

```
$image = \think\Image::open('./image.png');
// 按照原图的比例生成一个最大为150*150的缩略图并保存为thumb.png
$image->thumb(150,150,\think\Image::THUMB_SOUTHEAST)->save('./thumb.png')
;
```

生成的缩略图效果如图：

这里就不再对其他用法一一举例了。

图像翻转

使用`flip`可以对图像进行翻转操作，默认是以x轴进行翻转，例如：

```
$image = \think\Image::open('./image.png');
// 对图像进行以x轴进行翻转操作
```

```
$image->flip()->save('./filp_image.png');
```

生成的效果如图：

我们也可以改变参数，以y轴进行翻转，例如：

```
$image = \think\Image::open('./image.png');
// 对图像进行以y轴进行翻转操作
$image->flip(\think\image::FLIP_Y)->save('./filp_image.png');
```

生成的效果如图：

> 图像的翻转可以理解为图像的镜面效果与图像旋转有所不同。

图像旋转

使用 `rotate` 可以对图像进行旋转操作（默认是顺时针旋转90度），我们用默认90度进行旋转举例：

```
$image = \think\Image::open('./image.png');
// 对图像使用默认的顺时针旋转90度操作
$image->rotate()->save('./rotate_image.png');
```

生成的效果如图：

图像保存参数

`save` 方法可以配置的参数

参数	默认	描述

pathname	必填项	图像保存路径名称
type	默认与原图相同	图像类型
quality	80	图像质量
interlace	true	是否对JPEG类型图像设置隔行扫描

> 设置隔行扫描的情况下在网页进行浏览时。是从上到下一行一行的显示，否则图片整个显示出来 然后由模糊到清晰显示。

添加水印

系统支持添加图片及文字水印，下面依次举例说明

添加图片水印，我们下载官网logo文件到根目录进行举例：

```
$image = \think\Image::open('./image.png');
// 给原图左上角添加水印并保存water_image.png
$image->water('./logo.png')->save('water_image.png');
```

`water` 方法的第二个参数表示水印的位置，默认值是 `WATER_SOUTH` ，可以传入下列 `\think\Image` 类的常量或者对应的数字：

```
//常量，标识左上角水印
const WATER_NORTHWEST = 1;
//常量，标识上居中水印
const WATER_NORTH     = 2;
//常量，标识右上角水印
const WATER_NORTHEAST = 3;
//常量，标识左居中水印
const WATER_WEST      = 4;
//常量，标识居中水印
const WATER_CENTER    = 5;
//常量，标识右居中水印
const WATER_EAST      = 6;
//常量，标识左下角水印
const WATER_SOUTHWEST = 7;
//常量，标识下居中水印
const WATER_SOUTH     = 8;
//常量，标识右下角水印
const WATER_SOUTHEAST = 9;
```

我们用左上角来进行测试：

```
$image = \think\Image::open('./image.png');
// 给原图左上角添加水印并保存water_image.png
$image->water('./logo.png',\think\Image::WATER_NORTHWEST)->save('water_im
```

```
age.png');
```

生成的图片效果如下：

还可以支持水印图片的透明度（0~100，默认值是100），例如：

```
$image = \think\Image::open('./image.png');
// 给原图左上角添加透明度为50的水印并保存alpha_image.png
$image->water('./logo.png',\think\Image::WATER_NORTHWEST,50)->save('alpha
_image.png');
```

生成的图片效果如下：

也可以支持给图片添加文字水印（我们复制一个字体文件`HYQingKongTiJ.ttf`到入口目录），我们现在生成一个像素20px，颜色为`#ffffff`的水印效果：

```
$image = \think\Image::open('./image.png');
// 给原图左上角添加水印并保存water_image.png
$image->text('十年磨一剑 - 为API开发设计的高性能框架','HYQingKongTiJ.ttf',20,'
#ffffff')->save('text_image.png');
```

生成的图片效果：

文字水印参数

文字水印比较多，在此只做说明不做演示了

参数	默认	描述
text	不能为空	添加的文字
font	不能为空	字体文件路径
size	不能为空	字号，单位是像素
color	#00000000	文字颜色
locate	WATER_SOUTHEAST	文字写入位置
offset	0	文字相对当前位置的偏移量
angle	0	文字倾斜角度

Time

时间戳操作

首先通过 composer 安装

```
composer require topthink/think-helper
```

在文件头部引入

```
use think\helper\Time;
```

比如需要获得今天的零点时间戳和23点59分59秒的时间戳

```
list($start, $end) = Time::today();

echo $start; // 零点时间戳
echo $end; // 23点59分59秒的时间戳
```

完整示例如下:

```
// 今日开始和结束的时间戳
Time::today();

// 昨日开始和结束的时间戳
Time::yesterday();

// 本周开始和结束的时间戳
Time::week();

// 上周开始和结束的时间戳
Time::lastWeek();

// 本月开始和结束的时间戳
Time::month();

// 上月开始和结束的时间戳
Time::lastMonth();

// 今年开始和结束的时间戳
Time::year();

// 去年开始和结束的时间戳
Time::lastYear();
```

```
// 获取7天前零点到现在的时间戳
Time::dayToNow(7)

// 获取7天前零点到昨日结束的时间戳
Time::dayToNow(7, true)

// 获取7天前的时间戳
Time::daysAgo(7)

//  获取7天后的时间戳
Time::daysAfter(7)

// 天数转换成秒数
Time::daysToSecond(5)

// 周数转换成秒数
Time::weekToSecond(5)
```

数据库迁移工具

数据库迁移工具

首先通过 composer 安装

```
composer require topthink/think-migration
```

注意事项，不支持修改文件配置目录

在命令行下运行查看帮助，可以看到新增的命令

```
php think
```

```
 migrate
  migrate:create     Create a new migration
  migrate:rollback   Rollback the last or to a specific migration
  migrate:run        Migrate the database
  migrate:status     Show migration status
 optimize
  optimize:autoload  Optimizes PSR0 and PSR4 packages to be loaded wit
h classmaps too, good for production.
  optimize:config    Build config and common file cache.
  optimize:route     Build route cache.
  optimize:schema    Build database schema cache.
 seed
  seed:create        Create a new database seeder
  seed:run           Run database seeders
```

创建迁移类，首字母必须为大写

```
php think migrate:create Users
```

可以看到目录下有新文件 .\database\migrations\20161117144043_users.php

使用实例

```
<?php

use Phinx\Migration\AbstractMigration;
```

```
class Users extends AbstractMigration
{
    /**
     * Change Method.
     */
    public function change()
    {
        // create the table
        $table = $this->table('users',array('engine'=>'MyISAM'));
        $table->addColumn('username', 'string',array('limit' => 15,'defau
lt'=>'','comment'=>'用户名，登陆使用'))
            ->addColumn('password', 'string',array('limit' => 32,'default
'=>md5('123456'),'comment'=>'用户密码'))
            ->addColumn('login_status', 'boolean',array('limit' => 1,'def
ault'=>0,'comment'=>'登陆状态'))
            ->addColumn('login_code', 'string',array('limit' => 32,'defau
lt'=>0,'comment'=>'排他性登陆标识'))
            ->addColumn('last_login_ip', 'integer',array('limit' => 11,'d
efault'=>0,'comment'=>'最后登录IP'))
            ->addColumn('last_login_time', 'datetime',array('default'=>0,
'comment'=>'最后登录时间'))
            ->addColumn('is_delete', 'boolean',array('limit' => 1,'defaul
t'=>0,'comment'=>'删除状态，1已删除'))
            ->addIndex(array('username'), array('unique' => true))
            ->create();
    }

    /**
     * Migrate Up.
     */
    public function up()
    {

    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
}
```

对于同一个数据表，如果需要新的迁移动作，例如删除字段、创建字段，可以创建新的更改文件，像svn一样往前记录操作，方便回滚。

更具体的使用可查看

http://docs.phinx.org/en/latest/

Workerman

Workerman

Workerman是一款纯PHP开发的开源高性能的PHP socket 服务器框架。被广泛的用于手机app、手游服务端、网络游戏服务器、聊天室服务器、硬件通讯服务器、智能家居、车联网、物联网等领域的开发。 支持TCP长连接，支持Websocket、HTTP等协议，支持自定义协议。基于workerman开发者可以更专注于业务逻辑开发，不必再为PHP Socket底层开发而烦恼。

首先通过 composer 安装

```
composer require topthink/think-worker
```

如果需要在window下做服务端，还需要

```
composer require workerman/workerman-for-win
```

运行出现错误PHP Fatal error: Call to undefined function Workerman\Lib\pcntl_signal()，需要删除vendor\workerman\workerman，防止命名覆盖。

服务端使用示例如下:

在项目根目录新增启动服务文件server.php

```
#!/usr/bin/env php
<?php
namespace think;

define('APP_PATH', __DIR__ . '/application/');

// 加载基础文件
require __DIR__ . '/thinkphp/base.php';

// 执行应用并响应
Container::get('app',[APP_PATH])->bind('push/Worker')->run()->send();
```

新增服务处理类（`push.app`是本地测试域名）

```php
<?php

namespace app\push\controller;

use think\worker\Server;

class Worker extends Server
{
    protected $socket = 'websocket://push.app:2346';

    /**
     * 收到信息
     * @param $connection
     * @param $data
     */
    public function onMessage($connection, $data)
    {
        $connection->send('我收到你的信息了');
    }

    /**
     * 当连接建立时触发的回调函数
     * @param $connection
     */
    public function onConnect($connection)
    {

    }

    /**
     * 当连接断开时触发的回调函数
     * @param $connection
     */
    public function onClose($connection)
    {

    }

    /**
     * 当客户端的连接上发生错误时触发
     * @param $connection
     * @param $code
     * @param $msg
     */
    public function onError($connection, $code, $msg)
    {
        echo "error $code $msg\n";
    }

    /**
     * 每个进程启动
     * @param $worker
     */
    public function onWorkerStart($worker)
    {
```

```
    }
}
```

在命令行下运行，启动监听服务

```
php server.php
```

打开chrome浏览器，先打开 `push.app` 域名下的网页（js跨域不能通讯），按F12打开调试控制台，在Console一栏输入(或者把下面代码放入到html页面用js运行)

```
ws = new WebSocket("ws://push.app:2346");
ws.onopen = function() {
    alert("连接成功");
    ws.send('tom');
    alert("给服务端发送一个字符串：tom");
};
ws.onmessage = function(e) {
    alert("收到服务端的消息：" + e.data);
};
```

继续测试

```
ws.send('保持连接，发第二次信息，查看服务器回应');
```

MongoDb

使用`Mongo`之前，需要装PHP的`mongo`扩展，访问 http://pecl.php.net/package/mongodb ，选择最新的版本即可，然后选择你的PHP版本对应的扩展。

然后使用`Composer`安装扩展包：

```
composer require topthink/think-mongo
```

接下来，需要修改数据库配置文件中的相关参数：

```
    // 数据库类型
    'type'            => '\think\mongo\Connection',
    // 设置查询类
    'query'                     => '\think\mongo\Query',
    // 服务器地址
    'hostname'        => '127.0.0.1',
    // 集合名
    'database'        => 'demo',
    // 用户名
    'username'        => '',
    // 密码
    'password'        => '',
    // 端口
    'hostport'        => '',
```

默认安装的`mongodb`是没有用户名和密码的，可以留空。如果你的服务器安装的mongodb提供了用户名和密码认证，请自行修改。`MongoDb`一样支持分布式设置，设置方法和`Mysql`的分布式设置一致。

关于主键

`MongoDb`会自动添加`_id`字段而且作为主键，该主键数据是一个`MongoDB\BSON\ObjectID`对象实例。

为了方便查询，系统做了封装，该主键的值可以直接当作字符串使用，因此下面的查询是有效的：

```
// 查询操作
$user = Db::table('user')
```

```
        ->where('_id','589461c0fc122812b4007411')
    ->find();
// 或者直接使用
$user = Db::table('user')
    ->find('589461c0fc122812b4007411');
```

为了保持和Mysql一致的主键命名习惯，系统提供了一个数据库配置参数`pk_convert_id`可以强制把`_id`转换为`id`进行操作。

```
// 强制把_id转换为id
'pk_convert_id'  => true,
```

设置后，就可以把`id`当成`_id`来使用

```
// 查询操作
$user = Db::table('user')
        ->where('id','589461c0fc122812b4007411')
    ->find();
dump($user);
```

输出结果为：

```
array (size=3)
  'name' => string 'thinkphp' (length=8)
  'email' => string 'thinkphp@qq.com' (length=15)
  'id' => string '589461c0fc122812b4007411' (length=24)
```

原来的`_id`已经变成`id`，而且是一个字符串类型。

方法变化和差异

除了常规的CURD方法之外，包括`value`、`column`、`setInc`、`setDec`、`setField`、`paginate`等方法仍然被支持，更新的时候也支持`data`、`inc`和`dec`方法，包括聚合查询方法也都支持。

由于数据库自身的原因，以下链式方法在`MongoDb`中不再支持（或者无效）：

不再支持的方法
view
join
alias

|group|
|having|
|union|
|lock|
|strict|
|sequence|
|force|
|bind|
|partition|

针对了`MongoDb`的特殊性增加了如下链式操作方法：

方法	描述
skip	设置skip
awaitData	设置awaitData
batchSize	设置batchSize
exhaust	设置exhaust
modifiers	设置modifiers
noCursorTimeout	设置noCursorTimeout
oplogReplay	设置oplogReplay
partial	设置partial
maxTimeMS	设置maxTimeMS
slaveOk	设置slaveOk
tailable	设置tailable
writeConcern	设置writeConcern

并且`fetchPdo`方法改为`fetchCursor`。

Mongo原生查询

系统提供了几个基础查询方法，仅供熟悉`MongoDb`语法的用户使用。

query ($collection, $query)

collection：表示当前查询的集合
query：是一个`\MongoDB\Driver\Query`对象，详细用法可以参考官方手册

代码示例如下

```
$filter = [
    'author' => 'bjori',
    'views' => [
        '$gte' => 100,
    ],
];

$options = [
    /* Only return the following fields in the matching documents */
    'projection' => [
        'title' => 1,
        'article' => 1,
    ],
    /* Return the documents in descending order of views */
    'sort' => [
        'views' => -1
    ],
);

$query = new MongoDB\Driver\Query($filter, $options);
Db::query('demo.user', $query);
```

execute ($collection, $bulk)

collection：表示当前查询的集合
bulk：是一个 `\MongoDB\Driver\BulkWrite` 对象，详细用法可以参考官方手册

command ($command, $dbName)

command：是一个 `\MongoDB\Driver\Command` 对象，详细用法参考官方手册

dbName：当前操作的数据库名称，留空表示当前数据库

除此之外，系统还封装了一个 `cmd` 方法可以直接执行字符串格式的 `mongo` 命令，例如：

```
// 列出当前的集合
$collections = Db::cmd('listCollections');
```

更多的 `mongo` 命令参考 `MongoDb` 官方手册。

单元测试

单元测试

首先安装`ThinkPHP5`的单元测试扩展，进入命令行，切换到tp5的应用根目录下面，执行：

```
composer require topthink/think-testing
```

> 由于单元测试扩展的依赖较多，因此安装过程会比较久，请耐心等待。

安装完成后，会在应用根目录下面增加`tests`目录和`phpunit.xml`文件。
默认带了一个`tests/ExampleTest.php`单元测试文件，我们可以直接在命令行下面运行单元测试：

```
php think unit
```

> 请始终使用以上命令进行单元测试，而不是直接用 `phpunit` 来运行单元测试。

添加单元测试文件

我们来添加一个新的单元测试文件，单元测试文件为`tests/IndexTest.php`，内容如下：

```
<?php
use tests\TestCase;

class IndexTest extends TestCase
{

    public function testSomethingIsTrue()
    {
        $this->assertTrue(true);
    }

}
```

> 注意，单元测试文件中定义的测试类如果不存在冲突，可以不需要使用命名空间。

安全和性能

安全建议

输入安全

虽然ThinkPHP的底层安全防护比之前版本要强大不少，但永远不要相信用户提交的数据，建议务必遵守下面规则：

- 设置`public`目录为唯一对外访问目录，不要把资源文件放入应用目录；
- 开启表单令牌验证避免数据的重复提交，能起到`CSRF`防御作用；
- 使用框架提供的请求变量获取方法（Request类`param`方法及`input`助手函数）而不是原生系统变量获取用户输入数据；
- 对不同的应用需求设置`default_filter`过滤规则（默认没有任何过滤规则），常见的安全过滤函数包括`stripslashes`、`htmlentities`、`htmlspecialchars`和`strip_tags`等，请根据业务场景选择最合适的过滤方法；
- 使用验证类对业务数据设置必要的验证规则；
- 如果可能开启强制路由或者设置MISS路由规则，严格规范每个URL请求；

数据库安全

在确保用户请求的数据安全之后，数据库的安全隐患就已经很少了，因为数据操作默认使用了PDO预处理机制及自动参数绑定功能，请确保：

- 尽量少使用字符串查询条件，如果不得已的情况下 使用手动参数绑定功能；
- 不要让用户输入决定要查询或者写入的字段；
- 对于敏感数据在输出的时候使用`hidden`方法进行隐藏；
- 对于数据的写入操作应当做好权限检查工作；
- 写入数据严格使用`field`方法限制写入字段；
- 对于需要输出到页面的数据做好必要的`XSS`过滤；

上传

网站的上传功能也是一个非常容易被攻击的入口，所以对上传功能的安全检查是尤其必要的。

系统的`think\File`提供了文件上传的安全支持，包括对文件后缀、文件类型、文件大小以及上传图片文件的合法性检查，确保你已经在上传操作中启用了这些合法性检查。

其它的一些安全建议：

- 对所有公共的操作方法做必要的安全检查，防止用户通过URL直接调用；

- 不要缓存需要用户认证的页面；
- 对用户的上传文件，做必要的安全检查，例如上传路径和非法格式；
- 对于项目进行充分的测试，不要生成业务逻辑的安全隐患（这可能是最大的安全问题）；
- 最后一点，做好服务器的安全防护，安全问题的关键其实是找到你的最薄弱环节；

优化建议

架构及开发过程优化建议：

- 路由尽量使用域名路由或者路由分组；
- 在路由中进行验证和权限判断；
- 合理规划数据表字段类型及索引；
- 结合业务逻辑使用数据缓存，减少数据库压力；

在应用完成部署之后，建议对应用进行相关优化，包括：

- 如果开发过程中开启了调试模式的话，关闭调试模式（参考调试模式）；
- 通过命令行生成类库映射文件；
- 通过命令行生成配置缓存文件；
- 生成数据表字段缓存文件；

附录

助手函数

> 在5.1版本中，大部分的助手函数都可以归结为一个函数 app() ，因为5.1全面采用容器管理类的实例，而 app() 函数又是容器的“管家”。

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
abort	中断执行并发送HTTP状态码
action	调用控制器类的操作
app	快速获取容器中的实例 支持依赖注入
behavior	执行某个行为
bind	快速绑定对象实例
cache	缓存管理
call	调用反射执行callable 支持依赖注入
class_basename	获取类名(不包含命名空间)
class_uses_recursive	获取一个类里所有用到的trait
config	获取和设置配置参数
container	获取容器对象实例
controller	实例化控制器
cookie	Cookie管理
db	实例化数据库类
debug	调试时间和内存占用
dump	浏览器友好的变量输出
env	获取环境变量（ `V5.1.3+` ）
exception	抛出异常处理
halt	变量调试输出并中断执行
input	获取输入数据 支持默认值和过滤
json	JSON数据输出
jsonp	JSONP数据输出
lang	获取语言变量值
model	实例化Model

parse_name	字符串命名风格转换
redirect	重定向输出
request	实例化Request对象
response	实例化Response对象
route	注册路由规则（`V5.1.3+`）
session	Session管理
token	生成表单令牌输出
trace	记录日志信息
trait_uses_recursive	获取一个trait里所有引用到的trait
url	Url生成
validate	实例化验证器
view	渲染模板输出
widget	渲染输出Widget
xml	XML数据输出

> 核心框架不依赖任何助手函数，系统只是加载了助手函数文件，而且你可以在应用的公共函数文件（模块公共函数文件中重写无效）中重写上面这些助手函数。

上面这些内置的系统助手函数大部分方法都可以通过`app`助手函数完成调用，以进行缓存操作为例。

```
cache('name');
// 可以使用
app('cache')->get('name');
cache('name','value');
// 可以使用
app('cache')->set('name','value');
```

```
model('User');
可以使用
app()->model('User');
```

具体可以参考架构->容器和依赖注入。

> `V5.1.3+`版本开始，所有的助手函数都可以直接用于配置文件。

升级指导

从5.0升级到5.1

由于 `5.1` 版本很多用法不同于 `5.0` 版本，本篇内容帮助你更顺利的从 `5.0` 版本迁移到 `5.1` 版本。

> 如非必要，在建项目请勿盲目升级，5.0版本依然持续维护中。

命名空间调整

如果你自定义了应用类库的命名空间，需要改为设置环境变量 `APP_NAMESPACE` 而不是应用配置文件，如果你使用了 `.env` 配置文件，可以在里面添加：

```
APP_NAMESPACE = 你的应用类库根命名空间名
```

然后，检查你的应用类库中 `use` 或者调用的系统类库，如果使用了下面的系统类库（主要涉及的类库是 `5.0` 静态调用的系统类库），那么命名空间需要调整如下：

5.0系统	5.1系统
think\App	think\facade\App（或者 App）
think\Cache	think\facade\Cache（或者 Cache）
think\Config	think\facade\Config（或者 Config）
think\Cookie	think\facade\Cookie（或者 Cookie）
think\Debug	think\facade\Debug（或者 Debug）
think\Env	think\facade\Env（或者 Env）
think\Hook	think\facade\Hook（或者 Hook）
think\Lang	think\facade\Lang（或者 Lang）
think\Log	think\facade\Log（或者 Log）
think\Request	think\facade\Request（或者 Request）
think\Response	think\facade\Response（或者 Response）
think\Route	think\facade\Route（或者 Route）
think\Session	think\facade\Session（或者 Session）
think\Url	think\facade\Url（或者 Url）
think\Validate	think\facade\Validate（或者 Validate）
think\View	think\facade\View（或者 View）

> 如果只是用于依赖注入则无需更改命名空间。

后面括号里面的类名使用是的根命名空间（`\`），这是因为5.1对常用的系统核心类库做了类库别名，举个例子，如果应用类库开头`use`了`think\Url`

```
use think\Url;
Url::build('index/index');
```

则需要改成

```
use think\facade\Url;
Url::build('index/index');
```

或者

```
use Url;
Url::build('index/index');
```

> 5.1为系统的类库注册了类库别名，因此可以直接从根命名空间方式调用Url。

所以路由配置文件在迁移到`5.1`版本后你可以直接删除下面的一行代码

```
use think\Route;
```

配置文件调整

原有的配置文件`config.php`从应用目录移动到和应用目录同级的`config`目录，并拆分为`app.php`、`cache.php`等独立配置文件，系统默认的配置文件清单如下：

配置文件	说明
app.php	应用配置文件
cache.php	缓存配置文件
cookie.php	Cookie配置文件
database.php	数据库配置文件
log.php	日志配置文件
session.php	Session配置文件

template.php	模板引擎配置文件
trace.php	页面Trace配置文件

> 换而言之就是原来所有的一级配置都独立为一个配置文件

原来的应用 `extra` 目录下面的配置文件直接移动到 `config` 目录下面。
原来模块的配置文件（包括extra目录下面的）直接移动到模块下的 `config` 目录，然后参考上面的应用配置文件进行调整。

5.1的配置文件全部采用二级配置方式，所有不带一级配置名的参数都会作为 **`app`** 的二级配置，例如

```
config('app_debug');
```

等同于

```
config('app.app_debug');
```

> 并且注意，5.1的二级配置参数区分大小写。

一级配置 `app` 下的配置参数都在 `app.php` 配置文件中定义。

如果要获取数据库配置（ `database.php` 文件）的参数，则需要使用

```
config('database.hostname');
```

动态设置配置参数的时候，也要注意一级配置名

```
config('cache.type', 'memcache');
```

如果要获取一级配置下面的所有参数，使用

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

`view_replace_str` 配置参数改成template配置文件的 `tpl_replace_string` 配置参

数。

常量调整

`5.1`取消了所有的框架内置常量（不影响应用代码中的自定义常量），如需获取，请使用`think\facade\App`类的内置方法以及`think\facade\Env`类获取，下面给出的是`5.0`和`5.1`的常量对照表：

5.0常量	5.1获取方法
EXT	取消，固定使用`.php`
IS_WIN	取消
IS_CLI	取消
DS	使用PHP自带`DIRECTORY_SEPARATOR`
ENVPREFIX|取消，固定使用`PHP`	
THINK_START_TIME	`App::getBeginTime()`
THINK_START_MEM	`App::getBeginMem()`
THINK_VERSION	`App::version()`
THINK_PATH	`Env::get('think_path')`
LIB_PATH	`Env::get('think_path') . 'library/'`
CORE_PATH	`Env::get('think_path') . 'library/think/'`
APP_PATH	`Env::get('app_path')`
CONFIG_PATH	`Env::get('config_path')`
CONFIG_EXT	`App::getConfigExt()`
ROOT_PATH	`Env::get('root_path')`
EXTEND_PATH	`Env::get('root_path') . 'extend/'`
VENDOR_PATH	`Env::get('root_path') . 'vendor/'`
RUNTIME_PATH	`Env::get('runtime_path')`
LOG_PATH	`Env::get('runtime_path') . 'log/'`
CACHE_PATH	`Env::get('runtime_path') . 'cache/'`
TEMP_PATH	`Env::get('runtime_path'). 'temp/'`
MODULE_PATH	`Env::get('module_path')`

通过`Env`类的`get`方法获取路径变量的时候不区分大小写，例如下面的写法是等效的：

```
Env::get('root_path');
Env::get('ROOT_PATH');
```

路由调整

原有的路由定义文件`route.php`移动到应用目录同级的`route`目录下面，如果有定义其它的路由配置文件，一并放入`route`目录即可（无需更改文件名）。

`url_route_on`配置参数无效，会始终检查路由，没有定义路由的情况下默认解析方式依然有效。

原来的`before_behavior`和`after_behavior`参数更改为`before`和`after`，并且路由缓存功能暂时取消。

Route类的`rule`方法不再支持批量注册路由，请使用`Route::rules`方法替代。

如果使用了domain方法批量绑定模块，需要改为单独绑定，原来的用法：

```
Route::domain([
        'a' => 'a',
    'b' => 'b'
]);
```

需要改为：

```
Route::domain('a','a');
Route::domain('b','b');
```

数据库调整

- 取消了Query类的`getTableInfo`方法，可以用更加具体的`getTableFields`或者`getFieldsType`方法替代；
- 数据库查询后`5.1`不会清空查询条件；
- 取消了`select(false)`用法，使用`fetchSql()->select()`替代；
- 如果使用了mysql的JSON查询语法，`user$.name`需要改为`user->name`；
- 改变了查询构造器的数组多字段批量查询，从原来的

  ```
  where([
  'name'  =>      ['like','think%'],
    'id'  =>      ['>',0],
  ])
  ```

  需要调整为

  ```
  where([
  ```

```
['name','like','think%'],
  ['id','>',0],
])
```

或者使用表达式语法

```
where('name','like','think%')->where('id','>',0)
```

对于纯等于的数组条件则无需调整

```
where(['name'=>'think', 'type'=>1])
```

模型调整

为了确保模型的用法统一，对模型进行了一些调整，包括：

- 模型的数据集查询始终返回数据集对象而不再是数组；
- 模型的数据表主键如果不是`id`，则必须设置模型的`pk`属性；
- 软删除trait引入更改为`use think\model\concern\SoftDelete`；
- 全局查询范围`base`方法中无需添加软删除条件；
- 聚合模型功能废除，使用关联模型配合关联自动写入功能替代，更灵活；
- 模型的查询范围`scope`方法调用后只能使用数据库的查询方法；
- 取消模型的数据验证功能，请使用控制器验证或者路由验证替代；

控制器调整

为了规范化，继承了`think\Controller`类的话，初始化方法从原来的`_initialize`方法更改为`initialize`。

`fetch`方法以及`view`助手函数的`replace`参数废弃，如果需要模板替换功能，改成template配置文件的`tpl_replace_string`配置参数。或者使用`filter`方法进行过滤。

官方扩展

官方的下列`composer`扩展请升级到最新的`2.0`版本：

```
topthink/think-captcha
topthink/think-mongo
topthink/think-migration
topthink/think-testing
topthink/think-queue
```

其它注意事项

`Request` 类不再需要 `instance` 方法，直接调用类的方法即可。

废弃了 `Rest` 控制器扩展，建议更改为资源控制器的方式。
原来内置的其它控制器扩展，请自行在应用里面扩展。

因为严格遵循 `PSR-4` 规范，不再建议手动导入类库文件，所以新版取消了 `Loader::import` 方法以及 `import` 和 `vendor` 助手函数，推荐全面采用命名空间方式的类以及自动加载机制，如果必须使用请直接改为php内置的 `include` 或者 `require` 语法。

为了保持 `Loader` 类库的单纯性，原 `Loader` 类的 `controller` 、 `model` 、 `action` 和 `validate` 方法改为 `App` 类的同名方法，助手函数用法保持不变。

模板的变量输出默认添加了 `htmlentities` 安全过滤，如果你需要输出html内容的话，请使用 `{$var|raw}` 方式替换，并且 `date` 方法已经做了内部封装，无需再使用 `###` 变量替换了。

> 最后一个步骤不要忘了：清空缓存目录下的所有文件

更新日志

版本更新日志

V5.1.5 （2018-1-31）

该版本主要增强了数据库的JSON查询，并支持JSON字段的聚合查询，改进了一些性能问题，修正了路由的一些BUG，主要更新如下：

- 改进数据集查询对 `JSON` 数据的支持
- 改进聚合查询对 `JSON` 字段的支持
- 模型类增加 `getOrFail` 方法
- 改进数据库驱动的 `parseKey` 方法
- 改进Query类 `join` 方法的自关联查询
- 改进数据查询不存在不生成查询缓存
- 增加 `run` 命令行指令启动内置服务器
- `Request` 类 `pathinfo` 方法改进对 `cli-server` 支持
- `Session` 类增加 `use_lock` 配置参数设置是否启用锁机制
- 优化 `File` 缓存自动生成空目录的问题
- 域名及分组路由支持 `append` 方法传递隐式参数
- 改进日志的并发写入问题
- 改进 `Query` 类的 `where` 方法支持传入 `Query` 对象
- 支持设置单个日志文件的文件名
- 修正路由规则的域名条件约束
- `Request` 类增加 `subDomain` 方法用于获取当前子域名
- `Response` 类增加 `allowCache` 方法控制是否允许请求缓存

- `Request` 类增加 `sendData` 方法便于扩展
- 改进 `Env` 类不依赖 `putenv` 方法
- 改进控制台 `trace` 显示错误
- 改进 `MorphTo` 关联
- 改进完整路由匹配后带斜线访问出错的情况
- 改进路由的多级分组问题
- 路由url地址生成支持多级分组
- 改进路由Url生成的 `url_convert` 参数的影响
- 改进 `miss` 和 `auto` 路由内部解析
- 取消预载入关联查询缓存功能

V5.1.4 （2018-1-19）

该版本主要增强了数据库和模型操作，主要更新如下：

- 支持设置 `deleteTime` 属性为 `false` 关闭软删除
- 模型增加 `getError` 方法
- 改进Query类的 `getTableFields` / `getFieldsType` 方法 支持表名自动获取
- 模型类 `toCollection` 方法增加参数指定数据集类
- 改进 `union` 查询
- 关联预载入 `with` 方法增加缓存参数
- 改进模型类的 `get` 和 `all` 方法的缓存 支持关联缓存
- 支持 `order by field` 操作
- 改进 `insertAll` 分批写入
- 改进 `json` 字段数据支持
- 增加JSON数据的模型对象化操作
- 改进路由 `ext` 参数检测
- 修正 `rule` 方法的 `method` 参数使用 `get|post` 方式注册路由的问题

V5.1.3 （2018-1-12）

该版本主要改进了路由及调整函数加载顺序，主要更新如下：

- 增加 `env` 助手函数；
- 增加 `route` 助手函数;
- 增加视图路由方法；
- 增加路由重定向方法;
- 路由默认区分最后的目录斜杆（支持设置不区分）；

- 调整公共文件和配置文件的加载顺序（可以在配置文件中直接使用助手函数）；
- 视图类增加 `filter` 方法设置输出过滤；
- `view` 助手函数增加 `filter` 参数;
- 改进缓存生成指令；
- Session类的 `get` 方法支持获取多级；
- Request类 `only` 方法支持指定默认值;
- 改进路由分组;
- 修正使用闭包查询的时候自动数据缓存出错的情况;
- 废除 `view_filter` 钩子位置;
- 修正分组下面的资源路由;
- 改进session驱动;

V5.1.2（2018-1-8）

该版本改进了配置类及数据库类，主要更新如下：

- 修正嵌套路由分组；
- 修正自定义模板标签界定符后表达式语法出错的情况；
- 修正自关联的多次调用问题；
- 修正数组查询的 `null` 条件查询；
- 修正Query类的 `order` 及 `field` 的一处可能的BUG；
- 配置参数设置支持三级；
- 配置对象支持 `ArrayAccess` ；
- App类增加 `path` 方法用于设置应用目录；
- 关联定义增加 `selfRelation` 方法用于设置是否为自关联；

V5.1.1（2018-1-3）

修正一些反馈的BUG，包括：

- 修正Cookie类存取数组的问题
- 修正Controller的 `fetch` 方法
- 改进跨域请求
- 修正 `insertAll` 方法
- 修正 `chunk` 方法

V5.1.0（2018-1-1）

主要更新如下：

- 增加注解路由支持
- 路由支持跨域请求设置
- 增加 `app_dispatch` 钩子位置
- 修正多对多关联的 `detach` 方法
- 修正软删除的 `destroy` 方法
- Cookie类 `httponly` 参数默认为false
- 日志File驱动增加 `single` 参数配置记录同一个文件（不按日期生成）
- 路由的 `ext` 和 `denyExt` 方法支持不传任何参数
- 改进模型的 `save` 方法对 `oracle` 的支持
- Query类的 `insertall` 方法支持配合 `data` 和 `limit` 方法
- 增加 `whereOr` 动态查询支持
- 日志的ip地址记录改进
- 模型 `saveAll` 方法支持 `isUpdate` 方法
- 改进 `Pivot` 模型的实例化操作
- 改进Model类的 `data` 方法
- 改进多对多中间表模型类
- 模型增加 `force` 方法强制更新所有数据
- Hook类支持设置入口方法名称
- 改进验证类
- 改进 `hasWhere` 查询的数据重复问题
- 模型的 `saveall` 方法返回数据集对象
- 改进File缓存的 `clear` 方法
- 缓存添加统一的序列化机制
- 改进泛三级域名的绑定
- 改进泛域名的传值和取值
- Request类增加 `panDomain` 方法
- 改进废弃字段判断
- App类增加 `create` 方法用于实例化应用类库
- 容器类增加 `has` 方法
- 改进多数据库切换连接
- 改进断线重连的异常捕获
- 改进模型类 `buildQuery` 方法
- Query类增加 `unionAll` 方法
- 关联统计功能增强（支持Sum/Max/Min/Avg）

- 修正延迟写入
- chunk方法支持复合主键
- 改进JSON类型的写入
- 改进Mysql的insertAll方法
- Model类 `save` 方法改进复合主键包含自增的情况
- 改进Query类 `inc` 和 `dec` 方法的关键字处理
- File缓存inc和dec方法保持原来的有效期
- 改进redis缓存的有效期判断
- 增加checkRule方法用于单独数据的多个验证规则
- 修正setDec方法的延迟写入
- max和min方法增加force参数
- 二级配置参数区分大小写
- 改进join方法自关联的问题
- 修正关联模型自定义表名的情况
- Query类增加getFieldsType和getTableFields方法
- 取消视图替换功能及view_replace_str配置参数
- 改进域名绑定模块后的额外路由规则问题
- 改进mysql的insertAll方法
- 改进insertAll方法写入json字段数据的支持
- 改进redis长连接多编号库的情况

RC3版本（2017-11-6）

主要更新如下：

- 改进redis驱动的 `get` 方法
- 修正Query类的 `alias` 方法
- `File` 类错误信息支持多语言
- 修正路由的额外参数解析
- 改进 `whereTime` 方法
- 改进Model类 `getAttr` 方法
- 改进App类的 `controller` 和 `validate` 方法支持多层
- 改进 `HasManyThrough` 类
- 修正软删除的 `restore` 方法
- 改进 `MorpthTo` 关联
- 改进数据库驱动类的 `parseKey` 方法

- 增加 `whereField` 动态查询方法
- 模型增加废弃字段功能
- 改进路由的 `after` 行为检查和 `before` 行为机制
- 改进路由分组的检查
- 修正mysql的 `json` 字段查询
- 取消Connection类的 `quote` 方法
- 改进命令行的支持
- 验证信息支持多语言
- 修正路由模型绑定
- 改进参数绑定类型对枚举类型的支持
- 修正模板的 `{$Think.version}` 输出
- 改进模板 `date` 函数解析
- 改进 `insertAll` 方法支持分批执行
- Request类 `host` 方法支持反向代理
- 改进 `JumpResponse` 支持区分成功和错误模板
- 改进开启类库后缀后的关联外键自动识别问题
- 修正一对一关联的JOIN方式预载入查询问题
- Query类增加 `hidden` 方法

RC2版本（2017-10-17）

主要更新如下：

- 修正视图查询
- 修正资源路由
- 修正 `HasMany` 关联 修正 `where` 方法的闭包查询
- 一对一关联绑定属性到父模型后 关联属性不再保留
- 修正应用的命令行配置文件读取
- 改进 `Connection` 类的 `getCacheKey` 方法
- 改进文件上传的非法图像异常
- 改进验证类的 `unique` 规则
- Config类 `get` 方法支持获取一级配置
- 修正count方法对 `fetchSql` 的支持
- 修正mysql驱动对 `socket` 支持
- 改进Connection类的 `getRealSql` 方法
- 修正 `view` 助手函数

- Query类增加 `leftJoin` `rightJoin` 和 `fullJoin` 方法
- 改进app_namespace的获取
- 改进 `append` 方法对一对一 `bind` 属性的支持
- 改进关联的 `saveall` 方法的返回值
- 路由标识设置异常修复
- 改进Route类 `rule` 方法
- 改进模型的 `table` 属性设置
- 改进composer autofile的加载顺序
- 改进 `exception_handle` 配置对闭包的支持
- 改进app助手函数增加参数
- 改进composer的加载路径判断
- 修正路由组合变量的URL生成
- 修正路由URL生成
- 改进 `whereTime` 查询并支持扩展规则
- File类的 `move` 方法第二个参数支持 `false`
- 改进Config类
- 改进缓存类 `remember` 方法
- 惯例配置文件调整 Url类当普通模式参数的时候不做 `urlencode` 处理
- 取消 `ROOT_PATH` 和 `APP_PATH` 常量定义 如需更改应用目录 自己重新定义入口文件
- 增加 `app_debug` 的 `Env` 获取
- 修正泛域名绑定
- 改进查询表达式的解析机制
- mysql增加 `regexp` 查询表达式 支持正则查询
- 改进查询表达式的异常判断
- 改进model类的 `destroy` 方法
- 改进Builder类 取消 `parseValue` 方法
- 修正like查询的参数绑定问题
- console和start文件移出核心纳入应用库
- 改进Db类主键删除方法
- 改进泛域名绑定模块
- 取消 `BIND_MODULE` 常量 改为在入口文件使用 `bind` 方法设置
- 改进数组查询
- 改进模板渲染的异常处理
- 改进控制器基类的架构方法参数

- 改进Controller类的 `success` 和 `error` 方法
- 改进对浏览器 `JSON-Handle` 插件的支持
- 优化跳转模板的移动端显示
- 修正模型查询的 `chunk` 方法对时间字段的支持
- 改进trace驱动
- Collection类增加 `push` 方法
- 改进Redis Session驱动
- 增加JumpResponse驱动

RC1（2017-9-8）

主要新特性为：

- 引入容器和Facade支持
- 依赖注入完善和支持更多场景
- 重构的（对象化）路由
- 配置和路由目录独立
- 取消系统常量
- 助手函数增强
- 类库别名机制
- 模型和数据库增强
- 验证类增强
- 模板引擎改进
- 支持PSR-3日志规范
- RC1版本取消了5.0多个字段批量数组查询的方式