
ThinkPHP6.0
完全开发手册

# 目　录

# 序言

> 手册阅读须知：本手册仅针对ThinkPHP `6.0.*` 版本。

ThinkPHP®
中文WEB应用开发框架

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代码的同时，更注重易用性。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` 十四周年福利系列——助力开发 生态共赢


> ThinkPHP `6.0` 基于精简核心和统一用法两大原则在 `5.1` 的基础上对底层架构做了进一步的优化改进，并更加规范化。由于引入了一些新特性，ThinkPHP `6.0` 运行环境要求 `PHP7.1+` ，不支持 `5.1` 的无缝升级（官方给出了升级指导用于项目的升级参考）。

**`V6.0`** 版本由亿速云独家赞助发布

## 主要新特性

- 采用 `PHP7` 强类型（严格模式）
- 支持更多的 `PSR` 规范
- 多应用支持
- `ORM` 组件独立
- 改进的中间件机制
- 更强大和易用的查询
- 全新的事件系统
- 支持容器 `invoke` 回调
- 模板引擎组件独立

- 内部功能中间件化
- SESSION机制改进
- 缓存及日志支持多通道
- 引入 `Filesystem` 组件
- 对 `Swoole` 以及协程支持改进
- 对IDE更加友好
- 统一和精简大量用法

## 版权申明



## 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13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^_^。

# 基础

# 安装

ThinkPHP `6.0` 的环境要求如下：

> - PHP >= 7.1.0

> `6.0` 版本开始，必须通过 `Composer` 方式安装和更新，所以你无法通过 `Git` 下载安装。

## 安装 `Composer`

> 如果还没有安装 `Composer` ，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（英文文档，中文文档）。

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，我们建议使用国内镜像（阿里云）。

> 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://mirrors.aliyun.com/composer/
> ```

## 安装稳定版

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 tp
```

这里的 `tp` 目录名你可以任意更改，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 `app` 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。

## 安装开发版

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（适合学习过程），可以安装 `6.0.x-dev` 版本。

```
composer create-project topthink/think=6.0.x-dev tp
```

## 开启调试模式

应用默认是部署模式，在开发阶段，可以修改环境变量 `APP_DEBUG` 开启调试模式，上线部署后切换到部署模式。
本地开发的时候可以在应用根目录下面定义 `.env` 文件。

> 通过 `create-project` 安装后在根目录会自带一个 `.example.env` 文件（环境变量示例），你可以直接更名为 `.env` 文件并根据你的要求进行修改，该示例文件已经开启调试模式

## 测试运行

现在只需要做最后一步来验证是否正常运行。
进入命令行下面，执行下面指令

```
php think run
```

在浏览器中输入地址：

```
http://localhost:8000/
```

会看到欢迎页面。恭喜你，现在已经完成 `ThinkPHP6.0` 的安装！
如果你本地80端口没有被占用的话，也可以直接使用

```
php think run -p 80
```

然后就可以直接访问：

```
http://localhost/
```

> 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

`ThinkPHP6.0` 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（包含接口和Trait）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（包括接口和Trait）和文件名保持一致，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` ；
- 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

> 相对于 5.1 来说， 6.0 版本目录结构的主要变化是核心框架纳入 vendor 目录，然后原来的 application 目录变成 app 目录。

6.0 支持多应用模式部署，所以实际的目录结构取决于你采用的是单应用还是多应用模式，分别说明如下。

## 单应用模式

默认安装后的目录结构就是一个单应用模式

```
www  WEB部署目录（或者子目录）
├─app            应用目录
│  ├─controller      控制器目录
│  ├─model           模型目录
│  ├─ ...            更多类库目录
│  │
│  ├─common.php          公共函数文件
│  └─event.php           事件定义文件
│
├─config                 配置目录
│  ├─app.php             应用配置
│  ├─cache.php           缓存配置
│  ├─console.php         控制台配置
│  ├─cookie.php          Cookie配置
│  ├─database.php        数据库配置
│  ├─filesystem.php      文件磁盘配置
│  ├─lang.php            多语言配置
│  ├─log.php             日志配置
│  ├─middleware.php      中间件配置
│  ├─route.php           URL和路由配置
│  ├─session.php         Session配置
│  ├─trace.php           Trace配置
│  └─view.php            视图配置
│
├─view             视图目录
├─route                  路由定义目录
│  ├─route.php           路由定义文件
│  └─ ...
│
├─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
│
├─extend                 扩展类库目录
```

```
├─runtime                应用的运行时目录（可写，可定制）
├─vendor                 Composer类库目录
├─.example.env           环境变量示例文件
├─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
```

## 多应用模式

如果你需要一个多应用的项目架构，目录结构可以参考下面的结构进行调整（关于配置文件的详细结构参考后面章节）。

```
www  WEB部署目录（或者子目录）
├─app                应用目录
│  ├─app_name            应用目录
│  │  ├─common.php       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
│  │  ├─route            路由目录
│  │  └─ ...             更多类库目录
│  │
│  ├─common.php          公共函数文件
│  └─event.php           事件定义文件
│
├─config                 全局配置目录
│  ├─app.php             应用配置
│  ├─cache.php           缓存配置
│  ├─console.php         控制台配置
│  ├─cookie.php          Cookie配置
│  ├─database.php        数据库配置
│  ├─filesystem.php      文件磁盘配置
│  ├─lang.php            多语言配置
│  ├─log.php             日志配置
│  ├─middleware.php      中间件配置
│  ├─route.php           URL和路由配置
│  ├─session.php         Session配置
│  ├─trace.php           Trace配置
│  └─view.php            视图配置
│
├─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
│
├─extend                 扩展类库目录
```

```
├─runtime                应用的运行时目录（可写，可定制）
├─vendor                 Composer类库目录
├─.example.env           环境变量示例文件
├─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
```

> 多应用模式部署后，记得删除 `app` 目录下的 `controller` 目录（系统根据该目录作为判断是否单应用的依据）。

在实际的部署中，请确保只有 `public` 目录可以对外访问。

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

## 默认应用文件

默认安装后， `app` 目录下会包含下面的文件。

```
├─app                应用目录
│  │
│  ├─BaseController.php     默认基础控制器类
│  ├─ExceptionHandle.php    应用异常定义文件
│  ├─common.php             全局公共函数文件
│  ├─middleware.php         全局中间件定义文件
│  ├─provider.php           服务提供定义文件
│  ├─Request.php            应用请求对象
│  └─event.php              全局事件定义文件
```

`BaseController.php` 、 `Request.php` 和 `ExceptionHandle.php` 三个文件是系统默认提供的基础文件，位置你可以随意移动，但注意要同步调整类的命名空间。如果你不需要使用 `Request.php` 和 `ExceptionHandle.php` 文件，或者要调整类名，记得必须同步调整 `provider.php` 文件中的容器对象绑定。

> `provider.php` 服务提供定义文件只能全局定义，不支持在应用下单独定义

# 配置

## 配置目录

### 单应用模式

对于单应用模式来说，配置文件和目录很简单，根目录下的 `config` 目录下面就是所有的配置文件。每个配置文件对应不同的组件，当然你也可以增加自定义的配置文件。

```
├─config（配置目录）
│   ├─app.php             应用配置
│   ├─cache.php           缓存配置
│   ├─console.php         控制台配置
│   ├─cookie.php          Cookie配置
│   ├─database.php        数据库配置
│   ├─filesystem.php      文件磁盘配置
│   ├─lang.php            多语言配置
│   ├─log.php             日志配置
│   ├─middleware.php      中间件配置
│   ├─route.php           URL和路由配置
│   ├─session.php         Session配置
│   ├─trace.php           Trace配置
│   ├─view.php            视图配置
│   └─ ...                更多配置文件
│
```

单应用模式的 `config` 目录下的所有配置文件系统都会自动读取，不需要手动加载。如果存在子目录，你可以通过 `Config` 类的 `load` 方法手动加载，例如：

```
// 加载config/extra/config.php 配置文件 读取到extra
\think\facade\Config::load('extra/config', 'extra');
```

### 多应用模式

在多应用模式下，配置分为全局配置和应用配置。

- 全局配置： `config` 目录下面的文件就是项目的全局配置文件，对所有应用有效。
- 应用配置：每个应用可以有独立配置文件，相同的配置参数会覆盖全局配置。

```
├─app（应用目录）
│   ├─app1 （应用1）
│   │   └─config（应用配置）
│   │      ├─app.php              应用配置
│   │      ├─cache.php            缓存配置
```

```
│   │       ├─cookie.php          Cookie配置
│   │       ├─database.php        数据库配置
│   │       ├─lang.php            多语言配置
│   │       ├─log.php             日志配置
│   │       ├─route.php           URL和路由配置
│   │       ├─session.php         Session配置
│   │       ├─view.php            视图及模板引擎配置
│   │       ├─trace.php           Trace配置
│   │       └─ ...                更多配置文件
│   │
│   └─ app2...  （更多应用）
│
├─config（全局配置）
│   ├─app.php             应用配置
│   ├─cache.php           缓存配置
│   ├─console.php         控制台配置
│   ├─cookie.php          Cookie配置
│   ├─database.php        数据库配置
│   ├─filesystem.php      文件磁盘配置
│   ├─lang.php            多语言配置
│   ├─log.php             日志配置
│   ├─middleware.php      中间件配置
│   ├─route.php           URL和路由配置
│   ├─session.php         Session配置
│   ├─trace.php           Trace配置
│   ├─view.php            视图配置
│   └─ ...                更多配置文件
│
```

## 配置定义

可以直接在相应的全局或应用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。

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_debug | 应用调试模式 |
| config_ext | 配置文件后缀 |

### 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
```

> 默认安装后的根目录有一个 `.example.env` 环境变量示例文件，你可以直接改成 `.env` 文件后进行修改。

> 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 `off` ， `no` 和 `false` 等效于 布尔值 `false` ，值为 `yes` 、 `on` 和 `true` 等效于 布尔值的 `true` 。
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，可以使用

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

如果要设置一个没有键值的数组参数，可以使用

```
PATHINFO_PATH[] =  ORIG_PATH_INFO
PATHINFO_PATH[] =  REDIRECT_PATH_INFO
PATHINFO_PATH[] =  REDIRECT_URL
```

获取环境变量的值可以使用下面的方式获取：

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

要使用 `Env` 类，必须先引入 `think\facade\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'),
];
```

## 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，配置后全局或应用配置必须统一使用相同的配置类型。

# 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;
```

然后就可以使用下面的方法读取某个配置参数的值：

读取一级配置的所有参数（每个配置文件都是独立的一级配置）

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

读取单个配置参数

```
Config::get('app.app_name');
Config::get('route.url_domain_root');
```

读取数组配置（理论上支持无限级配置参数读取）

```
Config::get('database.default.host');
```

判断是否存在某个设置参数：

```
Config::has('template');
Config::has('route.route_rule_merge');
```

## 参数批量设置

`Config` 类不再支持动态设置某个配置参数，但可以支持批量设置更新配置参数。

```
// 批量设置参数
Config::set(['name1' => 'value1', 'name2' => 'value2'], 'config');
// 获取配置
Config::get('config');
```

## 系统配置文件

下面系统自带的配置文件列表及其作用：

| 配置文件名 | 描述 |
| --- | --- |
| app.php | 应用配置 |
| cache.php | 缓存配置 |
| console.php | 控制台配置 |
| cookie.php | Cookie配置 |
| database.php | 数据库配置 |
| filesystem.php | 磁盘配置 |
| lang.php | 多语言配置 |
| log.php | 日志配置 |
| middleware.php | 中间件配置 |
| route.php | 路由和URL配置 |
| session.php | Session配置 |
| trace.php | 页面Trace配置 |
| view.php | 视图配置 |

具体的配置参数及默认值可以直接查看应用 `config` 目录下面的相关文件内容。

> 如果是多应用模式的话配置文件可能同时存在全局和应用配置文件两个同名文件

## 使用 `Yaconf` 定义

可以支持使用 `Yaconf` 统一定义配置，但不支持动态设置。

> 安装了 `yaconf` 扩展之后，项目里面的配置文件不再有效。而且不再区分全局和应用配置。

首先需要安装 `topthink/think-yaconf` 扩展，

```
composer require topthink/think-yaconf
```

然后在 `app` 目录下的 `provider.php` 文件中添加：

```
'think\Config'  =>  'think\Yaconf',
```

使用 `setYaconf` 方法指定 `Yaconf` 使用的独立配置文件，例如：

```
// 建议在应用的公共函数文件中进行设置
think\facade\Config::setYaconf('thinkphp');
```

设置后，你只需要在 `thinkphp.ini` 一个文件里进行项目的配置，而无需分开多个文件，避免和其它项目冲突。

> 关于 `Yaconf` 的安装和配置用法可以参考这里。

# 架构

# 请求流程

## HTTP请求流程

对于一个HTTP应用来说，从用户发起请求到响应输出结束，大致的标准请求流程如下：

- 载入 `Composer` 的自动加载 `autoload` 文件
- 实例化系统应用基础类 `think\App`
- 获取应用目录等相关路径信息
- 加载全局的服务提供 `provider.php` 文件
- 设置容器实例及应用对象实例，确保当前容器对象唯一
- 从容器中获取 `HTTP` 应用类 `think\Http`
- 执行 `HTTP` 应用类的 `run` 方法启动一个 `HTTP` 应用
- 获取当前请求对象实例（默认为 `app\Request` 继承 `think\Request` ）保存到容器
- 执行 `think\App` 类的初始化方法 `initialize`
- 加载环境变量文件 `.env` 和全局初始化文件
- 加载全局公共文件、系统助手函数、全局配置文件、全局事件定义和全局服务定义
- 判断应用模式（调试或者部署模式）
- 监听 `AppInit` 事件
- 注册异常处理
- 服务注册
- 启动注册的服务
- 加载全局中间件定义
- 监听 `HttpRun` 事件
- 执行全局中间件
- 执行路由调度（ `Route` 类 `dispatch` 方法）
- 如果开启路由则检查路由缓存
- 加载路由定义
- 监听 `RouteLoaded` 事件
- 如果开启注解路由则检测注解路由
- 路由检测（中间流程很复杂 略）
- 路由调度对象 `think\route\Dispatch` 初始化
- 设置当前请求的控制器和操作名
- 注册路由中间件
- 绑定数据模型

- 设置路由额外参数
- 执行数据自动验证
- 执行路由调度子类的 `exec` 方法返回响应 `think\Response` 对象
- 获取当前请求的控制器对象实例
- 利用反射机制注册控制器中间件
- 执行控制器方法以及前后置中间件
- 执行当前响应对象的 `send` 方法输出
- 执行HTTP应用对象的 `end` 方法善后
- 监听 `HttpEnd` 事件
- 执行中间件的 `end` 回调
- 写入当前请求的日志信息

至此，当前请求流程结束。

# 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，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，负责处理请求（注意，不一定是HTTP请求）的生命周期，入口文件位于 `public` 目录下面，最常见的入口文件就是 `index.php` ， `6.0` 支持多应用多入口，你可以给每个应用增加入口文件，例如给后台应用单独设置的一个入口文件 `admin.php` 。
如果开启自动多应用的话，一般只需要一个入口文件 `index.php` 。

## 应用

`6.0` 版本提供了对多应用的良好支持，每个应用是一个 `app` 目录的子目录（或者指定的 `composer` 库），每个应用具有独立的路由、配置，以及MVC相关文件，这些应用可以公用框架核心以及扩展。而且可以支持 `composer` 应用加载。

## 容器

ThinkPHP使用（对象）容器统一管理对象实例及依赖注入。
容器类的工作由 `think\Container` 类完成，但大多数情况下我们都是通过应用类（ `think\App` 类）或是 `app` 助手函数来完成容器操作，容器中所有的对象实例都可以通过容器标识单例调用，你可以给容器中的对象实例绑定一个对象标识，如果没有绑定则使用类名作为容器标识。

## 系统服务

系统服务的概念是指在执行框架的某些组件或者功能的时候需要依赖的一些基础服务，服务类通常可以继承系统的 `think\Service` 类，但并不强制。
你可以在系统服务中注册一个对象到容器，或者对某些对象进行相关的依赖注入。由于系统服务的执行优先级问题，可以确保相关组件在执行的时候已经完成相关依赖注入。

## 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“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。

## 控制器

每个应用下面拥有独立的类库及配置文件，一个应用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（单应用模式）如下：

```
<?php
namespace app\controller;

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

> 一般建议继承一个基础的控制器，方便扩展。系统默认提供了一个 `app\BaseController` 控制器类。

## 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：

```
<?php
namespace app\controller;

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

```
    public function hello(string $name)
    {
        return 'Hello,'.$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，并且不是对象类型，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，操作方法的参数支持依赖注入。

## 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ThinkPHP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。
模型类通常需要继承 `think\Model` 类，一个典型的 `User` 模型器类如下：

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
}
```

## 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。
新版不再内置 `think-template` 模板引擎，如果需要使用ThinkPHP官方模板引擎，需要单独安装 `think-view` 模板引擎驱动扩展。

## 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。

> `6.0` 版本的驱动采用 `Composer` 的方式安装和管理。

## 中间件

中间件主要用于拦截或过滤应用的 `HTTP` 请求，并进行必要的业务处理。

> 新版部分核心功能使用中间件处理，你可以灵活关闭。包括Session功能、请求缓存和多语言功能。

## 事件

`6.0` 已经使用事件机制替代原来的行为和Hook机制，可以在应用中使用事件机制的特性来扩展功能。
此外数据库操作和模型操作在完成数据操作的回调机制，也使用了事件机制。

## 助手函数

系统为一些常用的操作提供了助手函数支持。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 `6.0` 采用单一入口模式进行项目部署和访问，一个应用都有一个统一（但不一定是唯一）的入口。如果采用自动多应用部署的话，一个入口文件还可以自动对应多个应用。

## 入口文件定义

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

```
// [ 应用入口文件 ]
namespace think;

require __DIR__ . '/../vendor/autoload.php';

// 执行HTTP应用并响应
$http = (new App())->http;
$response = $http->run();
$response->send();
$http->end($response);
```

> 如果你没有特殊的自定义需求，无需对入口文件做任何的更改。

> 入口文件位置的设计是为了让应用部署更安全，请尽量遵循 `public` 目录为唯一的 `web` 可访问目录，其他的文件都可以放到非WEB访问目录下面。

## 控制台入口文件

除了应用入口文件外，系统还提供了一个控制台入口文件，位于项目根目录的 `think` （注意该文件没有任何的后缀）。

该入口文件代码如下：

```
#!/usr/bin/env php
<?php
namespace think;

// 加载基础文件
require __DIR__ . '/vendor/autoload.php';

// 应用初始化
(new App())->console->run();
```

控制台入口文件用于执行控制台指令，例如：

```
php think version
```

> 系统内置了一些常用的控制台指令，如果你安装了额外的扩展，也会增加相应的控制台指令，都是通过该入口文件执行的。

# 多应用模式

## 多应用

默认安装后使用单应用模式部署，目录结构如下：

```
├─app 应用目录
│  ├─controller          控制器目录
│  ├─model               模型目录
│  ├─view                视图目录
│  └─ ...                更多类库目录
│
├─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
│
├─view                   视图目录
├─config                 应用配置目录
├─route                  路由定义目录
├─runtime                应用的运行时目录
```

> 单应用模式的优势是简单灵活，URL地址完全通过路由可控。配合路由分组功能可以实现类似多应用的灵活机制。

如果要使用多应用模式，你需要安装多应用模式扩展 `think-multi-app` 。

```
composer require topthink/think-multi-app
```

然后你的应用目录结构需要做如下调整，主要区别在 `app` 目录增加了应用子目录，然后配置文件和路由定义文件都纳入应用目录下。

```
├─app 应用目录
│  ├─index               主应用
│  │  ├─controller       控制器目录
│  │  ├─model            模型目录
│  │  ├─view             视图目录
│  │  ├─config           配置目录
│  │  ├─route            路由目录
│  │  └─ ...             更多类库目录
│  │
│  ├─admin               后台应用
│  │  ├─controller       控制器目录
```

```
|   |   ├─model           模型目录
|   |   ├─view            视图目录
|   |   ├─config          配置目录
|   |   ├─route           路由目录
|   |   └─ ...            更多类库目录
|
├─public                  WEB目录（对外访问目录）
|   ├─admin.php           后台入口文件
|   ├─index.php           入口文件
|   ├─router.php          快速测试文件
|   └─.htaccess           用于apache的重写
|
├─config                  全局应用配置目录
├─runtime                 运行时目录
|   ├─index               index应用运行时目录
|   └─admin               admin应用运行时目录
```

从目录结构可以看出来，每个应用相对保持独立，并且可以支持多个入口文件，应用下面还可以通过多级控制器来维护控制器分组。

## 自动多应用部署

支持在同一个入口文件中访问多个应用，并且支持应用的映射关系以及自定义。如果你通过 `index.php` 入口文件访问的话，并且没有设置应用 `name` ，系统自动采用自动多应用模式。
自动多应用模式的URL地址默认使用

```
// 访问admin应用
http://serverName/index.php/admin
// 访问shop应用
http://serverName/index.php/shop
```

> 也就是说 `pathinfo` 地址的第一个参数就表示当前的应用名，后面才是该应用的路由或者控制器/操作。

如果直接访问

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

访问的其实是 `index` 默认应用，可以通过 `app.php` 配置文件的 `default_app` 配置参数指定默认应用。

```
// 设置默认应用名称
'default_app' => 'home',
```

接着访问

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

其实访问的是 `home` 应用。

> 自动多应用模式下，路由是每个应用独立的，所以你没法省略URL里面的应用参数。但可以使用域名绑定解决。

## 多应用智能识别

如果没有绑定入口或者域名的情况下，URL里面的应用不存在，例如访问：

```
http://serverName/index.php/think
```

假设并不存在 `think` 应用，这个时候系统会自动切换到单应用模式，如果有定义全局的路由，也会进行路由匹配检查。
如果我们在 `route/route.php` 全局路由中定义了：

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

访问上面的URL就会输出

```
hello,ThinkPHP!
```

如果你希望 `think` 应用不存在的时候，直接访问默认应用的路由，可以在 `app.php` 中配置

```
// 开启应用快速访问
'app_express'    =>    true,
// 默认应用
'default_app'    =>    'home',
```

这个时候就会访问 `home` 应用下的路由。

## 增加应用入口

允许为每个应用创建单独的入口文件而不通过 `index.php` 入口文件访问多个应用，例如创建一个 `admin.php` 入口文件来访问 `admin` 应用。

```
// [ 应用入口文件 ]
namespace think;

require __DIR__ . '/../vendor/autoload.php';

// 执行HTTP应用并响应
$http = (new  App())->http;
$response = $http->run();
$response->send();
$http->end($response);
```

> 多应用使用不同的入口的情况下，每个入口文件的内容都是一样的，默认入口文件名（不含后缀）就是应用名。

使用下面的方式访问 `admin` 应用

```
http://serverName/admin.php
```

如果你的入口文件名和应用不一致，例如你的后台 `admin` 应用，入口文件名使用了 `test.php` ，那么入口文件需要改成：

```
// [ 应用入口文件 ]
namespace think;

require __DIR__ . '/../vendor/autoload.php';

// 执行HTTP应用并响应
$http = (new  App())->http;
$response = $http->name('admin')->run();
$response->send();
$http->end($response);
```

## 获取当前应用

如果需要获取当前的应用名，可以使用

```
app('http')->getName();
```

## 应用目录获取

单应用和多应用模式会影响一些系统路径的值，为了更好的理解本手册的内容，你可能需要理解下面几个系统路径所表示的位置。

| 目录位置 | 目录说明 | 获取方法（助手函数） |
| --- | --- | --- |
| 根目录 | 项目所在的目录，默认自动获取，可以在入口文件实例化 `App` 类的时候传入。 | `root_path()` |
| 基础目录 | 根目录下的 `app` 目录 | `base_path()` |
| 应用目录 | 当前应用所在的目录，如果是单应用模式则同基础目录，如果是多应用模式，则是 `app` /应用子目录 | `app_path()` |
| 配置目录 | 根目录下的 `config` 目录 | `config_path()` |
| 运行时目录 | 框架运行时的目录，单应用模式就是根目录的 `runtime` 目录，多应用模式为 `runtime` / 应用子目录 | `runtime_path()` |

> 注意：应用支持使用 `composer` 包，这个时候目录可能是 `composer` 包的类库所在目录。

对于非自动多应用部署的情况，如果要加载 `composer` 应用，需要在入口文件中设置应用路径：

```
// [ 应用入口文件 ]
namespace think;

require __DIR__ . '/../vendor/autoload.php';

// 执行HTTP应用并响应
$http = (new  App())->http;
$response = $http->path('path/to/app')->run();
$response->send();
$http->end($response);
```

## 应用映射

自动多应用模式下，支持应用的别名映射，例如：

```
'app_map' => [
    'think'  =>  'admin',  // 把admin应用映射为think
],
```

应用映射后，原来的应用名将不能被访问，例如上面的 `admin` 应用不能直接访问，只能通过 `think` 应用访问。

应用映射支持泛解析，例如：

```
'app_map' => [
    'think' =>  'admin',
    'home'  =>  'index',
    '*'     =>  'index',
],
```

表示如果URL访问的应用不在当前设置的映射里面，则自动映射为 `index` 应用。

如果要使用 `composer` 加载应用，需要设置

```
'app_map'    =>    [
    'think' => function($app) {
        $app->http->path('path/to/composer/app');
    },
],
```

## 域名绑定应用

如果你的多应用使用多个子域名或者独立域名访问，你可以在 `config/app.php` 配置文件中定义域名和应用的绑定。

```
'domain_bind' => [
    'blog'         =>  'blog',   //  blog子域名绑定到blog应用
    'shop.tp.com' =>  'shop',   //  完整域名绑定
    '*'            =>  'home', // 二级泛域名绑定到home应用
],
```

## 禁止应用访问

你如果不希望某个应用通过URL访问，例如，你增加了一个 `common` 子目录用于放置一些公共类库，你可以设置

```
'deny_app_list' =>    ['common']
```

# URL访问

## URL设计

`6.0` 的URL访问受路由影响，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）/控制器/操作/参数/值…
```

如果使用自动多应用模式的话，URL一般是

```
http://serverName/index.php/应用/控制器/操作/参数/值...
```

> 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

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?s=/控制器/操作/[参数名/参数值...]
```

## 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。

# 容器和依赖注入

## 容器和依赖注入

ThinkPHP使用容器来更方便的管理类依赖及运行依赖注入，新版的容器支持 `PSR-11` 规范。

> 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` 助手函数或者 `think\App` 类即可容器操作，如果在服务类中可以直接调用 `this->app` 进行容器操作。

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\controller;

use think\Request;

class Index
{
    protected $request;

    public function __construct(Request $request)
    {
        $this->request = $request;
    }

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

> 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；
- 路由的闭包定义；
- 事件类的执行方法；
- 中间件的执行方法；

对于自定义的类以及方法，如果需要使用依赖注入，需要使用系统提供的 `invoke` 助手函数调用，例如：

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}
```

如果直接 `new` 的话，需要手动传入 `Bar` 对象实例

```
$bar = new Bar();
$foo = new Foo($bar);
```

如果使用容器来实例化的话，可以自动进行依赖注入。

```
$foo = invoke('Foo');
```

如果要对某个方法支持依赖注入，可以使用

```
class Foo
{
    public function bar(Bar $bar)
    {
        // ...
    }
}
```

```
$result = invoke(['Foo', 'bar']);
```

也支持对某个函数或者闭包使用依赖注入

```
$result = invoke(function(Bar $bar) {
    // ...
});
```

## 绑定

依赖注入的类统一由容器进行管理，大多数情况下是在自动绑定并且实例化的。不过你可以随时进行手动绑定类到容器中（通常是在服务类的 `register` 方法中进行绑定），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
$this->app->bind('think\Cache', 'app\common\Cache');
```

或者使用助手函数

```
// 绑定类库标识
bind('cache', 'think\Cache');
```

> 绑定的类标识可以自己定义（只要不冲突）。

### 绑定闭包

可以绑定一个闭包到容器中

```
bind('sayHello', function ($name) {
    return 'hello,' . $name;
});
```

### 绑定实例

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);
```

### 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

在实际应用开发过程，不需要手动绑定，我们只需要在 `app` 目录下面定义 `provider.php` 文件（只能在全局定义，不支持应用单独定义），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\App | app |
| think\Cache | cache |
| think\Config | config |
| think\Cookie | cookie |
| think\Console | console |
| think\Db | db |
| think\Debug | debug |
| think\Env | env |
| think\Event | event |
| think\Http | http |
| think\Lang | lang |
| think\Log | log |
| think\Middleware | middleware |
| | |

| | |
|---|---|
| think\Request | request |
| think\Response | response |
| think\Filesystem | filesystem |
| think\Route | route |
| think\Session | session |
| think\Validate | validate |
| think\View | view |

## 解析

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

```
$cache = app('cache');
```

带参数实例化调用

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

对于没有绑定的类，也可以直接解析

```
$arrayItem = app('org\utils\ArrayItem');
```

> 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);
```

## 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;
```

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，但大多数情况下面，我们更建议使用依赖注入。

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

## 自动注入

容器主要用于依赖注入，依赖注入会首先检查容器中是否注册过该对象实例，如果没有就会自动实例化，然后自动注入，例如：

我们可以给路由绑定模型对象实例

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

class Index
{

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

}
```

## 自定义实例化

容器中的对象实例化支持自定义，可以在你需要依赖注入的对象中增加 `__make` 方法定义，例如：

如果你希望 `User` 模型类在依赖注入的时候 使用自定义实例化的方式，可以用下面的方法。

```
<?php
```

```
namespace app\index\model;

use think\Model;
use think\db\Query;

class User extends Model
{
    public static function __make(Query $query)
    {
        return (new self())->setQuery($query);
    }
}
```

## 容器对象回调机制

容器中的对象实例化之后，支持回调机制，利用该机制可以实现诸如注解功能等相关功能。

你可以通过 `resolving` 方法注册一个全局回调

```
Container::getInstance()->resolving(function($instance,$container) {
    // ...
});
```

回调方法支持两个参数，第一个参数是容器对象实例，第二个参数是容器实例本身。

或者单独注册一个某个容器对象的回调

```
Container::getInstance()->resolving(\think\Cache::class,function($instance,$con
tainer) {
    // ...
});
```

# 服务

## 系统服务

系统服务的概念是指在执行框架的某些组件或者功能的时候需要依赖的一些基础服务，服务类通常可以继承系统的 `think\Service` 类，但并不强制（如果继承 `think\Service` 的话可以直接调用 `this->app` 获取应用实例）。
你可以在系统服务中注册一个对象到容器，或者对某些对象进行相关的依赖注入。由于系统服务的执行优先级问题，可以确保相关组件在执行的时候已经完成相关依赖注入。

## 服务定义

你可以通过命令行生成一个服务类，例如：

```
php think make:service  FileSystemService
```

默认生成的服务类会继承系统的 `think\Service` ，并且自动生成了系统服务类最常用的两个空方法：`register` 和 `boot` 方法。

### 注册方法

`register` 方法通常用于注册系统服务，也就是将服务绑定到容器中，例如：

```
<?php
namespace app\service;

use my\util\FileSystem;

class FileSystemService extends Service
{
    public function register()
    {
        $this->app->bind('file_system', FileSystem::class);
    }
}
```

`register` 方法不需要任何的参数，如果你只是简单的绑定容器对象的话，可以直接使用 `bind` 属性。

```
<?php
namespace app\service;

use my\util\FileSystem;
```

```
class FileSystemService extends Service
{
    public $bind = [
        'file_system'    =>    FileSystem::class,
    ];
}
```

## 启动方法

`boot` 方法是在所有的系统服务注册完成之后调用，用于定义启动某个系统服务之前需要做的操作。例如：

```
<?php
namespace think\captcha;

use think\Route;
use think\Service;
use think\Validate;

class CaptchaService extends Service
{
    public function boot(Route $route)
    {
        $route->get('captcha/[:config]', "\\think\\captcha\\CaptchaController@index");

        Validate::maker(function ($validate) {
            $validate->extend('captcha', function ($value) {
                return captcha_check($value);
            }, ':attribute错误!');
        });
    }
}
```

`boot` 方法支持依赖注入，你可以直接使用其它的依赖服务。

## 服务注册

定义好系统服务后，你还需要注册服务到你的应用实例中。

可以在应用的全局公共文件 `service.php` 中定义需要注册的系统服务，系统会自动完成注册以及启动。例如：

```
return [
    '\app\service\ConfigService',
    '\app\service\CacheService',
];
```

如果你需要在你的扩展中注册系统服务，首先在扩展中增加一个服务类，然后在扩展的 `composer.json` 文件中增加如下定义：

```
"extra": {
    "think": {
        "services": [
            "think\\captcha\\CaptchaService"
        ]
    }
},
```

在安装扩展后会系统会自动执行 `service:discover` 指令用于生成服务列表，并在系统初始化过程中自动注册。

## 内置服务

为了更好的完成核心组件的单元测试，框架内置了一些系统服务类，主要都是用于核心类的依赖注入，包括 `ModelService` 、 `PaginatorService` 和 `ValidateService` 类。这些服务不需要注册，并且也不能卸载。

# 门面

## 门面（ `Facade` ）

门面为容器中的（动态）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。

## 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\Cache | think\facade\Cache |
| think\Config | think\facade\Config |
| think\Cookie | think\facade\Cookie |
| think\Db | think\facade\Db |
| think\Env | think\facade\Env |
| think\Event | think\facade\Event |
| think\Filesystem | think\facade\Filesystem |
| think\Lang | think\facade\Lang |
| think\Log | think\facade\Log |
| think\Middleware | think\facade\Middleware |
| think\Request | think\facade\Request |
| think\Response | think\facade\Response |
| think\Route | think\facade\Route |
| think\Session | think\facade\Session |
| 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');
```

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)
    {
    }
}
```

事实上，依赖注入和使用 `Facade` 代理的效果大多数情况下是一样的，都是从容器中获取对象实例。例如：

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

use think\Request;

class Index
{
    public function index(Request $request)
    {
        echo $request->controller();
    }
}
```

和下面的作用是一样的

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

use think\facade\Request;

class Index
{
    public function index()
    {
        echo Request::controller();
    }
}
```

依赖注入的优势是支持接口的注入，而 `Facade` 则无法完成。

> 一定要注意两种方式的 `use` 引入类库的区别

# 中间件

中间件主要用于拦截或过滤应用的 `HTTP` 请求，并进行必要的业务处理。

> 新版部分核心功能使用中间件处理，你可以灵活关闭。包括Session功能、请求缓存和多语言功能。

## 定义中间件

可以通过命令行指令快速生成中间件

```
php think make:middleware Check
```

这个指令会 `app/middleware` 目录下面生成一个 `Check` 中间件。

```
<?php

namespace app\middleware;

class Check
{
    public function handle($request, \Closure $next)
    {
        if ($request->param('name') == 'think') {
            return redirect('index/think');
        }

        return $next($request);
    }
}
```

中间件的入口执行方法必须是 `handle` 方法，而且第一个参数是 `Request` 对象，第二个参数是一个闭包。

> 中间件 `handle` 方法的返回值必须是一个 `Response` 对象。

在这个中间件中我们判断当前请求的 `name` 参数等于 `think` 的时候进行重定向处理。否则，请求将进一步传递到应用中。要让请求继续传递到应用程序中，只需使用 `$request` 作为参数去调用回调函数 `$next` 。

> 在某些需求下，可以使用第三个参数传入额外的参数。

```
<?php
```

```
namespace app\middleware;

class Check
{
    public function handle($request, \Closure $next, $name)
    {
        if ($name == 'think') {
            return redirect('index/think');
        }

        return $next($request);
    }
}
```

## 结束调度

中间件支持定义请求结束前的回调机制，你只需要在中间件类中添加 `end` 方法。

```
    public function end(\think\Response $response)
    {
        // 回调行为
    }
```

> 注意，在 `end` 方法里面不能有任何的响应输出。因为回调触发的时候请求响应输出已经完成了。

## 前置/后置中间件

中间件是在请求具体的操作之前还是之后执行，完全取决于中间件的定义本身。

下面是一个前置行为的中间件

```
<?php

namespace app\middleware;

class Before
{
    public function handle($request, \Closure $next)
    {
        // 添加中间件执行代码

        return $next($request);
    }
}
```

下面是一个后置行为的中间件

```php
<?php

namespace app\middleware;

class After
{
    public function handle($request, \Closure $next)
    {
        $response = $next($request);

        // 添加中间件执行代码

        return $response;
    }
}
```

中间件方法同样也可以支持依赖注入。

来个比较实际的例子，我们需要判断当前浏览器环境是在微信或支付宝

```php
namespace app\middleware;

/**
 * 访问环境检查，是否是微信或支付宝等
 */
class InAppCheck
{
    public function handle($request, \Closure $next)
    {
        if (preg_match('~micromessenger~i', $request->header('user-agent'))) {
            $request->InApp = 'WeChat';
        } else if (preg_match('~alipay~i', $request->header('user-agent'))) {
            $request->InApp = 'Alipay';
        }
        return $next($request);
    }
}
```

然后在你的移动版的应用里添加一个 `middleware.php` 文件

例如： `/path/app/mobile/middleware.php`

```php
return [
    app\middleware\InAppCheck::class,
];
```

然后在你的 `controller` 中可以通过 `request()->InApp` 获取相关的值

## 定义中间件别名

可以直接在应用配置目录下的 `middleware.php` 中先预定义中间件（其实就是增加别名标识），例如：

```
return [
    'alias' => [
        'auth'  => app\middleware\Auth::class,
        'check' => app\middleware\Check::class,
    ],
];
```

可以支持使用别名定义一组中间件，例如：

```
return [
    'alias' => [
        'check' => [
            app\middleware\Auth::class,
            app\middleware\Check::class,
        ],
    ],
];
```

## 注册中间件

新版的中间件分为全局中间件、应用中间件（多应用模式下有效）、路由中间件以及控制器中间件四个组。执行顺序分别为：

> 全局中间件->应用中间件->路由中间件->控制器中间件

### 全局中间件

全局中间件在 `app` 目录下面 `middleware.php` 文件中定义，使用下面的方式：

```
<?php

return [
    \app\middleware\Auth::class,
    'check',
    'Hello',
];
```

中间件的注册应该使用完整的类名，如果已经定义了中间件别名（或者分组）则可以直接使用。

全局中间件的执行顺序就是定义顺序。可以在定义全局中间件的时候传入中间件参数，支持两种方式传入。

```
<?php

return [
    [\app\http\middleware\Auth::class, 'admin'],
    'Check',
    ['hello','thinkphp'],
];
```

上面的定义表示 给 `Auth` 中间件传入 `admin` 参数，给 `Hello` 中间件传入 `thinkphp` 参数。

## 应用中间件

如果你使用了多应用模式，则支持应用中间件定义，你可以直接在应用目录下面增加 `middleware.php` 文件，定义方式和全局中间件定义一样，只是只会在该应用下面生效。

## 路由中间件

最常用的中间件注册方式是注册路由中间件

```
Route::rule('hello/:name','hello')
    ->middleware(\app\middleware\Auth::class);
```

支持注册多个中间件

```
Route::rule('hello/:name','hello')
    ->middleware([\app\middleware\Auth::class, \app\middleware\Check::class]);
```

然后，直接使用下面的方式注册中间件

```
Route::rule('hello/:name','hello')
    ->middleware('check');
```

支持对路由分组注册中间件

```
Route::group('hello', function(){
    Route::rule('hello/:name','hello');
})->middleware('auth');
```

支持对某个域名注册中间件

```
Route::domain('admin', function(){
    // 注册域名下的路由规则
```

```
})->middleware('auth');
```

如果需要传入额外参数给中间件，可以使用

```
Route::rule('hello/:name','hello')
    ->middleware('auth', 'admin');
```

如果需要定义多个中间件，使用数组方式

```
Route::rule('hello/:name','hello')
    ->middleware([Auth::class, 'Check']);
```

可以统一传入同一个额外参数

```
Route::rule('hello/:name','hello')
    ->middleware(['auth', 'check'], 'admin');
```

或者分开多次调用，指定不同的参数

```
Route::rule('hello/:name','hello')
    ->middleware('auth', 'admin')
        ->middleware('hello', 'thinkphp');
```

如果你希望某个路由中间件是全局执行（不管路由是否匹配），可以不需要在路由里面定义，支持直接在路由配置文件中定义，例如在 `config/route.php` 配置文件中添加：

```
'middleware'    =>    [
    app\middleware\Auth::class,
    app\middleware\Check::class,
],
```

这样，所有该应用下的请求都会执行 `Auth` 和 `Check` 中间件。

## 使用闭包定义中间件

你不一定要使用中间件类，在某些简单的场合你可以使用闭包定义中间件，但闭包函数必须返回 `Response` 对象实例。

```
Route::group('hello', function(){
    Route::rule('hello/:name','hello');
})->middleware(function($request,\Closure $next){
```

```
    if ($request->param('name') == 'think') {
        return redirect('index/think');
    }

    return $next($request);
});
```

## 控制器中间件

支持为控制器定义中间件，只需要在控制器中定义 `middleware` 属性，例如：

```
<?php
namespace app\controller;

class Index
{
    protected $middleware = ['auth'];

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

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

当执行 `index` 控制器的时候就会调用 `auth` 中间件，一样支持使用完整的命名空间定义。

如果需要设置控制器中间的生效操作，可以如下定义：

```
<?php
namespace app\controller;


class Index
{
    protected $middleware = [
        'auth'  => ['except'    => ['hello'] ],
        'check' => ['only'      => ['hello'] ],
    ];

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

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

## 中间件向控制器传参

可以通过给请求对象赋值的方式传参给控制器（或者其它地方），例如

```
<?php

namespace app\middleware;

class Hello
{
    public function handle($request, \Closure $next)
    {
        $request->hello = 'ThinkPHP';

        return $next($request);
    }
}
```

然后在控制器的方法里面可以直接使用

```
public function index(Request $request)
{
    return $request->hello; // ThinkPHP
}
```

## 执行优先级

如果对中间件的执行顺序有严格的要求，可以定义中间件的执行优先级。在配置文件中添加

```
return [
    'alias'    => [
        'check' => [
            app\middleware\Auth::class,
            app\middleware\Check::class,
        ],
    ],
    'priority' => [
        think\middleware\SessionInit::class,
        app\middleware\Auth::class,
```

```
        app\middleware\Check::class,
    ],
];
```

## 内置中间件

新版内置了几个系统中间件，包括：

| 中间件类 | 描述 |
| --- | --- |
| think\middleware\AllowCrossDomain | 跨域请求支持 |
| think\middleware\CheckRequestCache | 请求缓存 |
| think\middleware\LoadLangPack | 多语言加载 |
| think\middleware\SessionInit | Session初始化 |
| think\middleware\FormTokenCheck | 表单令牌 |

这些内置中间件默认都没有定义，你可以在应用的 `middleware.php` 文件中、路由或者控制器中定义这些中间件，如果不需要使用的话，取消定义即可。

# 事件

新版的事件系统可以看成是 `5.1` 版本行为系统的升级版，事件系统相比行为系统强大的地方在于事件本身可以是一个类，并且可以更好的支持事件订阅者。
事件相比较中间件的优势是事件比中间件更加精准定位（或者说粒度更细），并且更适合一些业务场景的扩展。例如，我们通常会遇到用户注册或者登录后需要做一系列操作，通过事件系统可以做到不侵入原有代码完成登录的操作扩展，降低系统的耦合性的同时，也降低了BUG的可能性。

> 事件系统的所有操作都通过 `think\facade\Event` 类进行静态调用

## 定义事件

事件系统使用了观察者模式，提供了解耦应用的更好方式。在你需要监听事件的位置，例如下面我们在用户完成登录操作之后添加如下事件触发代码：

```
// 触发UserLogin事件 用于执行用户登录后的一系列操作
Event::trigger('UserLogin');
```

或者使用助手函数

```
event('UserLogin');
```

这里 `UserLogin` 表示一个事件标识，如果你定义了单独的事件类，你可以使用事件类名（甚至可以传入一个事件类实例）。

```
// 直接使用事件类触发
event('app\event\UserLogin');
```

事件类可以通过命令行快速生成

```
php think make:event UserLogin
```

默认会生成一个 `app\event\UserLogin` 事件类，也可以指定完整类名生成。
我们可以给事件类添加方法

```
namespace app\event;

use app\model\User;
```

```
class UserLogin
{
    public $user;

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

一般事件类无需继承任何其它类。

你可以给事件类绑定一个事件标识，一般建议直接在应用的 `event.php` 事件定义文件中批量绑定。

```
return [
    'bind'     =>     [
        'UserLogin' => 'app\event\UserLogin',
        // 更多事件绑定
    ],
];
```

如果你需要动态绑定，可以使用

```
Event::bind(['UserLogin' => 'app\event\UserLogin']);
```

> ThinkPHP的事件系统不依赖事件类，如果没有额外的需求，仅通过事件标识也可以使用，省去定义事件类的麻烦。

如果你没有定义事件类的话，则无需绑定。对于大部分的场景，可能确实不需要定义事件类。

你可以在 `event` 方法中传入一个事件参数

```
// user是当前登录用户对象实例
event('UserLogin', $user);
```

如果是定义了事件类，可以直接传入事件对象实例

```
// user是当前登录用户对象实例
event(new UserLogin($user));
```

## 事件监听

你可以手动注册一个事件监听

```
Event::listen('UserLogin', function($user) {
    //
});
```

或者使用监听类来执行监听

```
Event::listen('UserLogin', 'app\listener\UserLogin');
```

可以通过命令行快速生成一个事件监听类

```
php think make:listener UserLogin
```

默认会生成一个 `app\listener\UserLogin` 事件监听类，也可以指定完整类名生成。

事件监听类只需要定义一个 `handle` 方法，支持依赖注入。

```
<?php
namespace app\listener;

class UserLogin
{
    public function handle($user)
    {
        // 事件监听处理
    }
}
```

在 `handle` 方法中如果返回了 `false` ，则表示监听中止，将不再执行该事件后面的监听。

一般建议直接在事件定义文件中定义对应事件的监听。

```
return [
    'bind'     =>    [
        'UserLogin' => 'app\event\UserLogin',
        // 更多事件绑定
    ],
    'listen'   =>    [
        'UserLogin'     =>    ['app\listener\UserLogin'],
        // 更多事件监听
    ],
];
```

## 事件订阅

可以通过事件订阅机制，在一个监听器中监听多个事件，例如通过命令行生成一个事件订阅者类，

```
php think make:subscribe User
```

默认会生成 `app\subscribe\User` 类，或者你可以指定完整类名生成。
然后你可以在事件订阅类中添加不同事件的监听方法，例如。

```
<?php
namespace app\subscribe;

class User
{
    public function onUserLogin($user)
    {
        // UserLogin事件响应处理
    }

    public function onUserLogout($user)
    {
        // UserLogout事件响应处理
    }
}
```

监听事件的方法命名规范是 `on` +事件标识（驼峰命名），如果希望统一添加事件前缀标识，可以定义 `eventPrefix` 属性。

```
<?php
namespace app\subscribe;

class User
{
    protected $eventPrefix = 'User';

    public function onLogin($user)
    {
        // UserLogin事件响应处理
    }

    public function onLogout($user)
    {
        // UserLogout事件响应处理
    }
}
```

如果希望自定义订阅方式（或者方法规范），可以定义 `subscribe` 方法实现。

```php
<?php
namespace app\subscribe;

use think\Event;

class User
{
    public function onUserLogin($user)
    {
        // UserLogin事件响应处理
    }

    public function onUserLogout($user)
    {
        // UserLogout事件响应处理
    }

    public function subscribe(Event $event)
    {
        $event->listen('UserLogin', [$this,'onUserLogin']);
        $event->listen('UserLogout',[$this,'onUserLogout']);
    }
}
```

然后在事件定义文件注册事件订阅者

```php
return [
    'bind'    =>    [
        'UserLogin' => 'app\event\UserLogin',
        // 更多事件绑定
    ],
    'listen'  =>    [
        'UserLogin'    =>    ['app\listener\UserLogin'],
        // 更多事件监听
    ],
    'subscribe'    =>    [
       'app\subscribe\User',
        // 更多事件订阅
    ],
];
```

如果需要动态注册，可以使用

```php
Event::subscribe('app\subscribe\User');
```

## 内置事件

内置的系统事件包括：

| 事件 | 描述 | 参数 |
| --- | --- | --- |
| AppInit | 应用初始化标签位 | 无 |
| HttpRun | 应用开始标签位 | 无 |
| HttpEnd | 应用结束标签位 | 当前响应对象实例 |
| LogWrite | 日志write方法标签位 | 当前写入的日志信息 |
| RouteLoaded | 路由加载完成 | 无 |

> `AppInit` 事件定义必须在全局事件定义文件中定义，其它事件支持在应用的事件定义文件中定义。

原来 `5.1` 的一些行为标签已经废弃，所有取消的标签都可以使用中间件更好的替代。可以把中间件看成处理请求以及响应输出相关的特殊事件。事实上，中间件的 `handler` 方法只是具有特殊的参数以及返回值而已。

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 |

> 查询事件的参数就是当前的查询对象实例。

模型事件包含：

| 钩子 | 对应操作 |
| --- | --- |
| after_read | 查询后 |
| before_insert | 新增前 |
| after_insert | 新增后 |
| before_update | 更新前 |
| after_update | 更新后 |
| before_write | 写入前 |
| after_write | 写入后 |
| before_delete | 删除前 |
| after_delete | 删除后 |

`before_write` 和 `after_write` 事件无论是新增还是更新都会执行。

> 模型事件方法的参数就是当前的模型对象实例。

# 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；
- 路由中间件支持；

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；
- 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；

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。

## 路由定义文件

路由规则的注册必须在应用的路由定义文件中完成。路由定义和检测是针对应用的，因此如果你采用的是多应用模式，每个应用的路由都是完全独立的，并且路由地址不能跨应用（除非采用重定向路由）。

> `route` 目录下的任何路由定义文件都是有效的，分开多个路由定义文件并没有实际的意义，纯粹出于管理方便而已。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。

```
├─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
```

如果你使用了多应用模式，那么路由定义文件则放入应用目录下：

```
├─app              应用目录
│  ├─app_name                应用目录
```

```
|   |   ├─common.php      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
|   |   ├─route           路由目录
|   |   |   ├─route.php   路由定义
|   |   |   ├─api.php     路由定义
|   |   |   └─...         更多路由定义
```

多应用模式下面，如果你开启了自动多应用，路由的规则是指在URL地址的应用名之后的部分，也就是说URL中的应用名是不能省略和改变的，例如你在 `index` 应用中定义了路由。

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

在没有开启自动多应用的情况下，URL地址是

```
http://serverName/index.php/hello/think
```

一旦你开启了自动多应用，那么实际的URL地址应该是

```
http://serverName/index.php/index/hello/think
```

> 如果不做特殊说明的话，后面章节的例子都采用单应用模式或者多个入口应用举例，如果你是自动多应用的话请参考上面的实例进行URL地址调整。

## 路由配置文件

路由的配置文件独立为 `config` 目录下的 `route.php` ，如果是多应用模式则支持在应用配置的 `route.php` 设置，请注意路由配置文件和路由定义文件之间的区别。

## 关闭路由

如果你的某个应用不需要使用路由功能，那么可以在应用的 `app.php` 配置文件中设置：

```
// 关闭应用的路由功能
'with_route'    =>    false,
```

关闭某个应用的路由。路由关闭后，你只能使用默认的URL解析规则来访问。

# 路由定义

要使用 `Route` 类注册路由必须首先在路由定义文件开头添加引用（后面不再重复说明）

```
use think\facade\Route;
```

## 注册路由

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');

例如注册如下路由规则（假设为单应用模式）：

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

我们访问：

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

会自动路由到：

```
http://serverName/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');
```

不过通常我们更推荐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。

## 规则表达式

规则表达式通常包含静态规则和动态规则，以及两种规则的结合，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）。

每个参数中可以包括动态变量，例如 `:变量` 或者 `<变量>` 都表示动态变量（新版推荐使用第二种方式，更利于混合变量定义），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。

## 可选变量

支持对路由参数的可选定义，例如：

```
Route::get('blog/:year/[:month]','Blog/archive');
// 或者
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地址开头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，可以在路由配置文件中设置

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

开启全局完全匹配后，如果需要对某个路由关闭完全匹配，可以使用

```
Route::get('new/:cate', 'News/category')->completeMatch(false);
```

## 额外参数

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

```
Route::get('blog/:id','blog/read')
    ->append(['status' => 1, 'app_id' =>5]);
```

上面的路由规则定义中 `status` 和 `app_id` 参数都是URL里面不存在的，属于隐式传值。可以针对不同的路由设置不同的额外参数。

> 如果 `append` 方法中的变量和路由规则存在冲突的话，append方法传入的优先。

## 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News控制器的read操作
Route::rule('new/:id','News/read')
    ->name('new_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，系统会默认使用路由地址作为路由标识，例如可以使用下面的方式生成

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

## 强制路由

在路由配置文件中设置

```
'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!';
});
```

# 变量规则

## 变量规则

> 系统默认的变量规则设置是 `\w+` ，只会匹配字母、数字、中文和下划线字符，并不会匹配特殊符号以及其它字符，需要定义变量规则或者调整默认变量规则。

可以在路由配置文件中自定义默认的变量规则，例如增加中划线字符的匹配：

```
'default_route_pattern' =>  '[\w\-]+',
```

支持在规则路由中指定变量规则，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 `$` ，也不支持模式修饰符，系统会自动添加。

### 全局变量规则

设置全局变量规则，全部路由有效：

```
// 支持批量添加
Route::pattern([
    'name' => '\w+',
    'id'   => '\d+',
]);
```

## 组合变量

如果你的路由规则比较特殊，可以在路由定义的时候使用组合变量。

例如：

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

## 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实际地址（或者响应对象）以及一些需要的额外参数，支持下面几种方式定义：

## 路由到控制器/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控制器和操作，然后由系统调度执行相关的操作，格式为：

> 控制器/操作

解析规则是从操作开始解析，然后解析控制器，例如：

```
// 路由到blog控制器
Route::get('blog/:id','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','group.Blog/read');
```

表示路由到下面的控制器类，

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

还可以支持路由到动态的应用、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', 'Blog/:action');
```

## 路由到类的方法

这种方式的路由可以支持执行任何类的方法，而不局限于执行控制器的操作方法。

路由地址的格式为（动态方法）：

> \完整类名@方法名

或者（静态方法）

> \完整类名::方法名

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');
```

## 重定向路由

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

支持路由直接渲染模板输出。

```
// 路由到模板文件
Route::view('hello/:name', 'index/hello');
```

表示该路由会渲染当前应用下面的 `view/index/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}!
```

## 路由到响应对象

支持在路由中直接指定响应对象输出，例如：

```
Route::get('hello/:name', response()
    ->data('Hello,ThinkPHP')
    ->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));
```

## 路由到闭包

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;
});
```

# 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后续行为。

> 路由参数可以在定义路由规则的时候直接传入（批量），推荐使用方法配置更加清晰。

| 参数 | 说明 | 方法名 |
| --- | --- | --- |
| 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 |
| complete_match | 是否完整匹配路由 | completeMatch |
| model | 绑定模型 | model |
| cache | 请求缓存 | cache |
| ajax | Ajax检测 | ajax |
| pjax | Pjax检测 | pjax |
| json | JSON检测 | json |
| validate | 绑定验证器类进行数据验证 | validate |
| append | 追加额外的参数 | append |
| middleware | 注册路由中间件 | middleware |
| filter | 请求变量过滤 | filter |

用法举例：

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

> 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。

如果你需要批量设置路由参数，也可以使用 `option` 方法。

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

## URL后缀

> URL后缀如果是全局统一的话，可以在路由配置文件中设置 `url_html_suffix` 参数，如果当前访问的URL地址中的URL后缀是允许的伪静态后缀，那么后缀本身是不会被作为参数值传入的。

不同参数设置的区别如下：

| 配置值 | 描述 |
| --- | --- |
| `false` | 禁止伪静态访问 |
| 空字符串 | 允许任意伪静态后缀 |
| `html` | 只允许设置的伪静态后缀 |
| `html\|htm` | 允许多个伪静态后缀 |

```
// 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();
```

### `AJAX` / `PJAX` / `JSON` 检测

可以检测当前是否为 `AJAX` / `PJAX` / `JSON` 请求。

```
// 必须是JSON请求访问
Route::get('new/:id', 'News/read')
    ->json();
```

### 请求变量检测

可以在匹配路由地址之外，额外检查请求变量是否匹配，只有指定的请求变量也一致的情况下才能匹配该路由。

```
// 检查type变量
Route::post('new/:id', 'News/save')
    ->filter('type', 1);

// 检查多个请求变量
Route::post('new/:id', 'News/save')
    ->filter([ 'type' => 1,'status'=> 1 ]);
```

### 追加额外参数

可以在定义路由的时候隐式追加额外的参数，这些参数不会出现在URL地址中。

```
Route::get('blog/:id', 'Blog/read')
    ->append(['app_id' => 1, 'status' => 1]);
```

在路由请求的时候会同时传入 `app_id` 和 `status` 两个参数。

## 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/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。

如果你的模型绑定使用的是 `id` 作为查询条件的话，还可以简化成下面的方式

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

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/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/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/hello')
    ->model(function ($id) {
        $model = new \app\index\model\User;
        return $model->where('id', $id)->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。

## 请求缓存

可以对当前的路由请求进行请求缓存处理，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。

## 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。

# 路由中间件

## 路由中间件

可以使用路由中间件，注册方式如下：

```
Route::rule('hello/:name','hello')
    ->middleware(\app\middleware\Auth::class);
```

或者对路由分组注册中间件

```
Route::group('hello', function(){
    Route::rule('hello/:name','hello');
})->middleware(\app\middleware\Auth::class);
```

如果需要传入额外参数给中间件，可以使用

```
Route::rule('hello/:name','hello')
    ->middleware(\app\middleware\Auth::class,'admin');
```

如果需要定义多个中间件，使用数组方式

```
Route::rule('hello/:name','hello')
    ->middleware([\app\middleware\Auth::class,\app\middleware\Check::class]);
```

可以统一传入同一个额外参数

```
Route::rule('hello/:name','hello')
    ->middleware([\app\middleware\Auth::class, \app\middleware\Check::class], '
admin');
```

如果你希望某个路由中间件是全局执行（不管路由是否匹配），可以不需要在路由里面定义，支持直接在路由配置文件中定义，例如在 `config/route.php` 配置文件中添加：

```
'middleware'    =>    [
    app\middleware\Auth::class,
    app\middleware\Check::class,
],
```

这样，所有该应用下的请求都会执行 `Auth` 和 `Check` 中间件。

更多中间件的用法参考架构章节的中间件内容。

# 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，例如：

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

分组路由支持所有的路由参数设置，具体参数的用法请参考路由参数章节内容。

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（也称之为虚拟分组），也可以使用：

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

路由分组支持嵌套，例如：

```
Route::group(function () {
    Route::group('blog', function () {
        Route::rule(':id', 'blog/read');
        Route::rule(':name', 'blog/read');
    });
})->ext('html')->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。

可以使用 `prefix` 方法简化相同路由地址的定义，例如下面的定义

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

可以简化为

```
Route::group('blog', function () {
    Route::get(':id', 'read');
    Route::post(':id', 'update');
    Route::delete(':id', 'delete');
})->prefix('blog/')->ext('html')->pattern(['id' => '\d+']);
```

## 路由完全匹配

如果希望某个分组下面的路由都采用完全匹配，可以使用

```
Route::group('blog', function () {
    Route::get(':id', 'read');
    Route::post(':id', 'update');
    Route::delete(':id', 'delete');
})->completeMatch()->prefix('blog/')->ext('html')->pattern(['id' => '\d+']);
```

## 延迟路由解析

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，你可以在路由配置文件中设置：

```
// 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。

> 通过路由分组或者域名路由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。

## 路由规则合并解析

同一个路由分组下的路由规则支持合并解析，而不需要遍历该路由分组下的所有路由规则，可以大大提升路由解析的性能。
对某个分组单独开启合并规则解析的用法如下：

```
Route::group('user', function () {
    Route::rule('hello/:name','hello');
    Route::rule('think/:name','think');
})->mergeRuleRegex();
```

这样该分组下的所有路由规则无论定义多少个都只需要匹配检查一次即可（实际上只会合并检查符合当前请求类型的路由规则）。

> `mergeRuleRegex` 方法只能用于路由分组或者域名路由（域名路由其实是一个特殊的分组）。

或者在路由配置文件中设置开启全局合并规则（对所有分组有效）

```
// 开启路由合并解析
'route_rule_merge'  => true,
```

## 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。

# 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', 'Blog');
```

表示注册了一个名称为 `blog` 的资源路由到 `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，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\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', 'Blog')
    ->vars(['blog' => 'blog_id']);
```

控制器的方法定义需要调整如下：

```
<?php
namespace app\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', 'Blog')
    ->only(['index', 'read', 'edit', 'update']);

// 排除index和delete操作
Route::resource('blog', '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', 'Blog');
Route::resource('blog.comment','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\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\controller;

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

# 注解路由

## 注解路由

ThinkPHP支持使用注解方式定义路由（也称为注解路由），如果需要使用注解路由需要安装额外的扩展：

```
composer require topthink/think-annotation
```

然后只需要直接在控制器类的方法注释中定义，例如：

```
<?php
namespace app\controller;

use think\annotation\Route;

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

`@Route("hello/:name")` 就是注解路由的内容，请务必注意注释的规范，不能在注解路由里面使用单引号，否则可能导致注解路由解析失败，可以利用IDE生成规范的注释。如果你使用 `PHPStorm` 的话，建议安装 `PHP Annotations` 插件：https://plugins.jetbrains.com/plugin/7320-php-annotations ，可以支持注解的自动完成。

> 该方式定义的路由在调试模式下面实时生效，部署模式则在第一次访问的时候生成注解缓存。

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\controller;

use think\annotation\Route;

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

如果有路由参数需要定义，可以直接在后面添加方法，例如：

```
<?php
namespace app\controller;

use think\annotation\Route;

class Index
{
    /**
     * @param string $name 数据名称
     * @Route('hello/:name', method="GET", https=1, ext="html")
     * @return mixed
     */
    public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

支持在类的注释里面定义资源路由，例如：

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

```
use think\annotation\route\Resource;

/**
 * @Resource("blog")
 */
class Blog
{
    public function index()
    {
    }

    public function read($id)
    {
    }

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

如果需要定义路由分组，可以使用

```
<?php
namespace app\controller;

use think\annotation\route\Group;
use think\annotation\route\Route;

/**
 * @Group("blog")
 */
class Blog
{
    /**
     * @param  string $name 数据名称
     * @return mixed
     * @Route("hello/:name", method="GET")
     */
    public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

当前控制器中的注解路由会自动加入 `blog` 分组下面，最终，会注册一个 `blog/hello/:name` 的路由规则。你一样可以对该路由分组设置公共的参数，例如：

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

use think\annotation\route\Middleware;
use think\annotation\route\Group;
use think\annotation\route\Route;
use think\middleware\SessionInit;

/**
 * @Group("blog",ext="html")
 * @Middleware({SessionInit::class})
 */
class Blog
{
    /**
     * @param  string $name 数据名称
     * @return mixed
     * @Route("hello/:name",method="GET")
     */
    public function hello($name)
    {
        return 'hello,'.$name;
    }
}
```

# 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：

## 绑定到控制器/操作

把当前的URL绑定到控制器/操作，最多支持绑定到操作级别，例如在路由定义文件中添加：

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

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

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

可以简化成

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

如果定义了路由

```
Route::get('blog/:id','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。

# 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', 'news/read');
    Route::rule(':user', 'user/info');
});
```

一旦定义了域名路由，该域名的访问就只会读取域名路由定义的路由规则。

> 闭包中可以使用路由的其它方法，包括路由分组，但不能再包含域名路由

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', 'news/read');
    Route::rule(':user', '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', 'news/read');
    Route::rule(':user', 'user/info');
})->crossDomainRule();
```

## 域名绑定

### 绑定到控制器类

```
// blog子域名绑定控制器
Route::domain('blog', '@blog');
```

### 绑定到命名空间

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

### 绑定到类

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

## 绑定到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');
})->bind('blog');
```

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

## 路由参数

域名路由本身也是一个路由分组，所以可以和路由分组一样定义公共的路由参数，例如：

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

# 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');
```

或者使用闭包定义

```
Route::miss(function() {
    return '404 Not Found!';
});
```

> 一旦设置了MISS路由，相当于开启了强制路由模式

当所有已经定义的路由规则都不匹配的话，会路由到 `miss` 方法定义的路由地址。

你可以限制 `MISS` 路由的请求类型

```
// 只有GET请求下MISS路由有效
Route::miss('public/miss', 'get');
```

## 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+']);
```

## 域名MISS路由

支持给某个域名设置单独的 `MISS` 路由

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

```
    Route::miss('blog/miss');
});
```

# 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-Modified-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')
    ->allowCrossDomain([
        'Access-Control-Allow-Origin'        => 'thinkphp.cn',
        'Access-Control-Allow-Credentials'   => 'true'
    ]);
```

# 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\Route::buildUrl()` 方法即可。该方法会返回一个 `think\route\Url` 对象实例，因为使用了 `__toString` 方法，因此可以直接输出路由地址。

```
echo \think\facade\Route::buildUrl();
```

如果是通过数据返回客户端，你可以先强制转换为字符串类型后再返回。

```
$url = (string) \think\facade\Route::buildUrl();
```

## 使用路由标识

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','blog/read');
```

在没有指定路由标识的情况下，可以直接使用路由地址来生成URL地址：

```
Route::buildUrl('blog/read', ['id' => 5, 'name' => 'thinkphp']);
```

如果我们在注册路由的时候指定了路由标识

```
Route::rule('blog/:id','blog/read')->name('blog_read');
```

那么必须使用路由标识来生成URL地址

```
Route::buildUrl('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
```

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' , 'blog/read');
```

可以使用下面的方式直接使用路由规则生成URL地址：

```
Route::buildUrl('/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：

```
Route::buildUrl('blog/read', ['id'=>5])->suffix('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

```
Route::buildUrl('index/blog/read',  ['id'=>5])
    ->suffix('shtml')
    ->domain(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：

```
Route::buildUrl('blog/read', ['id'=>5])->domain('blog');
```

或者传入完整的域名

```
Route::buildUrl('index/blog/read', ['id'=>5])->domain('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：

```
Route::buildUrl('index/blog/read@blog',  ['id'=>5]);
Route::buildUrl('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：

```
Route::buildUrl('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
```

## 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，可以使用下面的方法：

```
Route::buildUrl('index/blog/read', ['id'=>5])->root('/index.php');
```

## 助手函数

系统提供了一个 `url` 助手函数用于完成相同的功能，例如：

```
url('index/blog/read', ['id'=>5])
    ->suffix('html')
    ->domain(true)
    ->root('/index.php');
```

# 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 的控制器定义比较灵活，可以无需继承任何的基础类，也或者根据业务需求封装自己的基础控制器类。

# 控制器定义

## 控制器定义

控制器文件通常放在 `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）。

如果要改变 `controller` 目录名，需要在 `route.php` 配置文件中设置：

```
'controller_layer'    =>    'controllers',
```

如果使用的是单应用模式，那么控制器的类的定义如下：

```
<?php
namespace app\controller;

class User
{
    public function login()
    {
        return 'login';
    }
}
```

控制器类文件的实际位置则变成

```
app\controller\User.php
```

访问URL地址是（假设没有定义路由的情况下）

```
http://localhost/user/login
```

如果你的控制器是 `HelloWorld` ，并且定义如下：

```
<?php
namespace app\controller;

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

控制器类文件的实际位置是

```
app\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/HelloWorld/hello
```

并且也可以支持下面的访问URL

```
http://localhost/hello_world/hello
```

## 多应用模式

多应用模式下，控制器类定义仅仅是命名空间有所区别，例如：

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

class User
{
    public function login()
    {
        return 'login';
    }
}
```

控制器类文件的实际位置是

```
app\shop\controller\User.php
```

访问URL地址是（假设没有定义路由的情况下）

```
http://localhost/index.php/shop/user/login
```

## 控制器后缀

如果你希望避免引入同名模型类的时候冲突，可以在 `route.php` 配置文件中设置

```
// 使用控制器后缀
```

```
'controller_suffix'     => true,
```

这样，上面的控制器类就需要改成

```
<?php
namespace app\controller;

class UserController
{
    public function login()
    {
        return 'login';
    }
}
```

相应的控制器类文件也要改为

```
app\controller\UserController.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($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。并且控制器在 `json` 请求会自动转换为 `json` 格式输出。

> 不要在控制器中使用包括 `die` 、 `exit` 在内的中断代码。如果你需要调试并中止执行，可以使用系统提供的 `halt` 助手函数。

```
halt('输出测试');
```

## 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;

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

}
```

该控制器类的文件位置为：

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

访问地址可以使用

```
http://serverName/index.php/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：

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

# 基础控制器

大多数情况下，我们建议给你的控制器继承一个基础控制器。

默认安装后，系统提供了一个 `app\BaseController` 基础控制器类，你可以对该基础控制器进行修改。

> 基础控制器的位置可以随意放置，只需要注意更改命名空间即可。

该基础控制器仅仅提供了控制器验证功能，并注入了 `think\App` 和 `think\Request` 对象，因此你可以直接在控制器中使用 `app` 和 `request` 属性调用 `think\App` 和 `think\Request` 对象实例，下面是一个例子：

```
namespace app\controller;

use app\BaseController;

class Index extends BaseController
{
    public function index()
    {
        $action = $this->request->action();
        $path = $this->app->getBasePath();
    }
}
```

## 控制器验证

基础控制器提供了数据验证功能，使用如下：

```
namespace app\controller;

use app\BaseController;
use think\exception\ValidateException;

class Index extends BaseController
{
    public function index()
    {
        try {
            $this->validate( [
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ],  'app\index\validate\User');
        } catch (ValidateException $e) {
            // 验证失败 输出错误信息
```

```
            dump($e->getError());
        }
    }
}
```

> 该示例使用了验证器功能，具体可以参考验证章节的验证器部分，这里暂时不做展开。

如果需要批量验证，可以改为：

```
namespace app\controller;

use app\BaseController;
use think\exception\ValidateException;

class Index extends BaseController
{
    // 开启批量验证
    protected $batchValidate = true;

    public function index()
    {
        try {
            $this->validate( [
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ],  'app\index\validate\User');
        } catch (ValidateException $e) {
            // 验证失败 输出错误信息
            dump($e->getError());
        }
    }
}
```

# 空控制器

## 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，系统会尝试定位当前应用下的空控制器( `Error` )类，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。
例如，下面是单应用模式下，我们可以给项目定义一个 `Error` 控制器类。

```
<?php
namespace app\controller;

class Error
{
    public function __call($method, $args)
    {
        return 'error request!';
    }
}
```

# 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应用的Blog资源控制器使用：

```
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
```

如果只是用于接口开发，可以使用

```
php think make:controller index@Blog --api
```

然后你只需要为资源控制器注册一个资源路由：

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

设置后会自动注册7个路由规则，对应资源控制器的7个方法，更多内容请参考资源路由章节。

# 控制器中间件

## 控制器中间件

支持为控制器定义中间件，你只需要在你的控制器中定义 `middleware` 属性，例如：

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

use app\middleware\Auth;

class Index
{
    protected $middleware = [Auth::class];

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

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

当执行 `index` 控制器的时候就会调用 `Auth` 中间件，一样支持使用完整的命名空间定义。

如果需要设置控制器中间的生效操作，可以如下定义：

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

class Index
{
    protected $middleware = [
        Auth::class . ':admin'  => ['except'    => ['hello'] ],
        'Hello' => ['only'      => ['hello'] ],
    ];

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

    public function hello()
    {
```

```
        return 'hello';
    }
}
```

## 中间件传参

如果需要给中间件传参，可以的定义的时候使用

```
<?php
namespace app\controller;

class Index
{
    protected $middleware = ['Auth'];

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

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

## 控制器传参

可以通过给请求对象赋值的方式传参给控制器（或者其它地方），例如

```
<?php

namespace app\http\middleware;

class Hello
{
    public function handle($request, \Closure $next)
    {
        $request->hello = 'ThinkPHP';

        return $next($request);
    }
}
```

然后在控制器的方法里面可以直接使用

```
public function index(Request $request)
{
    return $request->hello; // ThinkPHP
}
```

# 请求

# 请求对象

当前的请求对象由 `think\Request` 类负责，该类不需要单独实例化调用，通常使用依赖注入即可。在其它场合则可以使用 `think\facade\Request` 静态类操作。

> 项目里面应该使用 `app\Request` 对象，该对象继承了系统的 `think\Request` 对象，但可以增加自定义方法或者覆盖已有方法。项目里面已经在 `provider.php` 中进行了定义，所以你仍然可以和之前一样直接使用容器和静态代理操作请求对象。

## 构造方法注入

一般适用于没有继承系统的控制器类的情况。

```
<?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');
    }
}
```

## 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\Request;

class Index
{

    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，请参考依赖注入章节。

## 静态调用

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\facade\Request;

class Index
{

    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;


class Index
{

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

## 自定义请求对象

你可以在项目里面自定义Request对象，修改已有的方法或者增加新的方法，默认已经在项目里面为你准备了 `app\Request` 类，你只需要直接修改该类就可以为你的项目单独自定义请求对象。
自定义请求对象不支持为多应用的某个应用自定义，只能是全局自定义，如果你需要为某个应用定义不同的请求对象，可以在入口文件里面修改。例如：

```
// 执行HTTP应用并响应
$request = new app\common\Request();
$http = (new App())->http;
$response = $http->run($request);
$response->send();
$http->end($response);
```

# 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 |
| `rootDomain` | 当前访问的根域名 |
| `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） |
| `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 |
| `rule` | 当前请求的路由对象实例 |

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

## 获取当前控制器/操作

可以通过请求对象获取当前请求的控制器/操作名。

| 方法 | 含义 |
| --- | --- |
| `controller` | 当前请求的控制器名 |
| `action` | 当前请求的操作名 |

获取当前控制器

```
Request::controller();
```

返回的是控制器的驼峰形式（首字母大写），和控制器类名保持一致（不含后缀）。

如果需要返回小写可以使用

```
Request::controller(true);
```

如果要返回小写+下划线的方式，可以使用

```
parse_name(Request::controller());
```

获取当前操作

```
Request::action();
```

返回的是当前操作方法的实际名称，如果需要返回小写可以使用

```
Request::action(true);
```

如果要返回小写+下划线的方式，可以使用

```
parse_name(Request::action());
```

如果使用了多应用模式，可以通过下面的方法来获取当前应用

```
app('http')->getName();
```

# 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，包括 `get/post/put/request/cookie/server/session/env/file` 。

## 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 |
| middleware | 获取 中间件赋值/传递的变量 |
| 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();
// 获取当前请求未经过滤的所有变量
Request::param(false);
// 获取部分变量
Request::param(['name', 'email']);
```

> `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和 `param` 方法用法基本一致。

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
```

> 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），其它变量名区分大小写。

## 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 `app\Request` 对象中设置 `filter` 全局过滤属性：

```
namespace app;

class Request extends \think\Request
{
    protected $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htmlspecialchars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\Filter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
```

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);
```

> 对于body中提交的 `json` 对象，你无需使用 `php://input` 去获取，可以直接当做表单提交的数据使用，因为系统已经自动处理过了

## 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']);
```

> 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。

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

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']);
```

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');

支持的变量修饰符，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');
```

## 中间件变量

可以在中间件里面设置和获取请求变量的值，这个值的改变不会影响 `PARAM` 变量的获取。

```
<?php

namespace app\http\middleware;
```

```
class Check
{
    public function handle($request, \Closure $next)
    {
        if ('think' == $request->name) {
            $request->name = 'ThinkPHP';
        }

        return $next($request);
    }
}
```

## 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htmlspecialchars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\Filter类的s
```

```
afeHtml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JSON请求 | isJson |
| 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，可以修改自定义Request类的 `varMethod` 属性：

## 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，可以修改自定义Request类的 `varAjax` 和 `varPjax` 属性：

> `_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');
```

# 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，我们在 `route.php` 中设置

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

的话，我们可以把下面的URL

```
http://serverName/blog/read/id/1
```

变成

```
http://serverName/blog/read/id/1.shtml
```

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

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/blog/3.html
http://serverName/blog/3.shtml
http://serverName/blog/3.xml
http://serverName/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/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/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

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。

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\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/blog/read/id/5
http://serverName/index.php/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/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：

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

那么会抛出下面的异常提示： `参数错误:id`

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

```
public function read($id = 0)
{
    return 'id=' . $id;
}
```

这样，当我们访问 `http://serverName/index.php/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）

为了更好的配合前端规范，支持自动识别小写+下划线的请求变量使用驼峰注入，例如：

```
http://serverName/index.php/blog/read/blog_id/5
```

可以使用下面的方式接收 `blog_id` 变量，所以请确保在方法的参数使用驼峰（首字母小写）规范。

```
public function read($blogId = 0)
{
    return 'id=' . $blogId;
}
```

# 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）。

```
// 定义GET请求路由规则 并关闭请求缓存（即使开启了全局请求缓存）
Route::get('new/:id','News/read')->cache(false);
```

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。

## 全局请求缓存

如果需要开启全局请求缓存，只需要在全局（或者应用）的中间件定义文件 `middleware.php` 中增加

```
    'think\middleware\CheckRequestCache',
```

然后只需要在 `route.php` 配置文件中设置全局缓存的有效时间（秒）：

```
'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_key` 参数，例如：

```
'request_cache_key' =>  '__URL__',
'request_cache_expire'  =>  3600,
```

缓存标识支持下面的特殊定义

| 标识 | 含义 |
| --- | --- |
| `__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_key'         => 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。

`ThinkPHP` 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。

> `Response` 类不能直接实例化，必须使用 `Response::make()` 静态方式创建，建议直接使用系统提供的助手函数完成。

# 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。

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\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。如果你发起一个JSON请求的话，输出就会自动使用JSON格式响应输出。

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 |
| 附件下载 | download | \think\response\Download |

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\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\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。

```
<?php
namespace app\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-revalidate'
]);
```

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。

## 写入Cookie

```
response()->cookie('name', 'value', 600);
```

## 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：

```
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

支持使用 `allowCache` 方法动态控制是否需要使用请求缓存。

```php
// 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\controller;

class Index
{
    public function hello()
    {
        return redirect('http://www.thinkphp.cn');
    }
}
```

### 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/hello/name/thinkphp');
```

如果你需要自动生成URL地址，应该在调用之前调用url函数先生成最终的URL地址。

```
redirect((string) url('hello',['name' => 'think']));
```

还可以支持使用 `with` 方法附加 `Session` 闪存数据重定向。

```
<?php
namespace app\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\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();
    }
```

```
}
```

# 文件下载

## 文件下载

支持文件下载功能，可以更简单的读取文件进行下载操作，支持直接下载输出内容。

你可以在控制器的操作方法中添加如下代码：

```
public function download()
    {
        // download是系统封装的一个助手函数
        return download('image.jpg', 'my.jpg');
    }
```

访问 `download` 操作就会下载命名为 `my.jpg` 的图像文件。

> 下载文件的路径是服务器路径而不是URL路径，如果要下载的文件不存在，系统会抛出异常。

下载文件名可以省略后缀，会自动判断（后面的代码都以助手函数为例）

```
public function download()
    {
        // 和上面的下载文件名是一样的效果
        return download('image.jpg', 'my');
    }
```

如果需要设置文件下载的有效期，可以使用

```
public function download()
    {
        // 设置300秒有效期
        return download('image.jpg', 'my')->expire(300);
    }
```

除了 `expire` 方法外，还支持下面的方法：

| 方法 | 描述 |
| --- | --- |
| name | 命名下载文件 |
| expire | 下载有效期 |
| isContent | 是否为内容下载 |
| mimeType | 设置文件的mimeType类型 |

助手函数提供了内容下载的参数，如果需要直接下载内容，可以在第三个参数传入 `true` ：

```
public function download()
{
    $data = '这是一个测试文件';
    return download($data, 'test.txt', true);
}
```

# 数据库

> 新版的数据库和模型操作已经独立为 `ThinkORM` 库，默认安装应用的时候会自动安装，如果你不需要使用该 `ORM` 库的话，可以单独卸载 `topthink/think-orm` 后安装其它的 `ORM` 库。

## 主要特性：

- 基于PDO和PHP强类型实现
- 支持原生查询和查询构造器
- 自动参数绑定和预查询
- 简洁易用的查询功能
- 强大灵活的模型用法
- 支持预载入关联查询和延迟关联查询
- 支持多数据库及动态切换
- 支持 `MongoDb`
- 支持分布式及事务
- 支持断点重连
- 支持JSON查询
- 支持数据库日志
- 支持 `PSR-16` 缓存及 `PSR-3` 日志规范

> 要使用Db类必须使用门面方式（ `think\facade\Db` ）调用

## 开发指南

关于 `ThinkORM` 更多的内容可以参考 `ThinkORM` 开发指南文档。

# 连接数据库

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

在全局或者应用配置目录（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 [
    'default'     =>    'mysql',
    'connections'     =>    [
        'mysql'     =>    [
            // 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_',
        ],
    ],
];
```

新版采用多类型的方式配置，方便切换数据库。

`default` 配置用于设置默认使用的数据库连接配置。

`connections` 配置具体的数据库连接信息， `default` 配置参数定义的连接配置必须要存在。

`type` 参数用于指定数据库类型

| type | 数据库 |
| --- | --- |
| mysql | MySQL |
| sqlite | SqLite |
| | |

| | |
|---|---|
| pgsql | PostgreSQL |
| sqlsrv | SqlServer |
| mongo | MongoDb |
| oracle | Oracle |

每个应用可以设置独立的数据库连接参数，通常直接更改 `default` 参数即可：

```
return [
    'default'    =>    'admin',
];
```

### 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。

## 切换连接

我们可以在数据库配置文件中定义多个连接信息

```
return [
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 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_',
        ],
        'demo'    =>    [
            // 数据库类型
            'type'        => 'mysql',
            // 服务器地址
            'hostname'    => '127.0.0.1',
            // 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'    => '',
            // 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_',
        ],
    ],
];
```

我们可以调用 `Db::connect` 方法动态配置数据库连接信息，例如：

```
\think\facade\Db::connect('demo')
    ->table('user')
    ->find();
```

> `connect` 方法必须在查询的最开始调用，而且必须紧跟着调用查询方法，否则可能会导致部分查询失效或

> 者依然使用默认的数据库连接。

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
{
    protected $connection = 'demo';
}
```

> 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。

## 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 |
|  | 数据库部署方式:0 集中 |  |

| | | |
|---|---|---|
| deploy | 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 |
| fields_cache | 是否开启字段缓存 | false |
| schema_cache_path | 字段缓存目录 | 无 |
| trigger_sql | 是否开启SQL监听 | true |
| auto_timestamp | 自动写入时间戳字段 | 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。

如果同时定义了 参数化数据库连接信息 和 dsn信息，则会优先使用dsn信息。

> 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。

## 断线重连

如果你使用的是长连接或者命令行，在超出一定时间后，数据库连接会断开，这个时候你需要开启断线重连才能确保应用不中断。

在数据库连接配置中设置：

```
// 开启断线重连
'break_reconnect' => true,
```

开启后，系统会自动判断数据库断线并尝试重新连接。大多数情况下都能自动识别，如果在一些特殊的情况下或者某些数据库驱动的断线标识错误还没有定义，支持配置下面的信息：

```
// 断线标识字符串
'break_match_str' => [
    'error with',
],
```

在 `break_match_str` 配置中加入你的数据库错误信息关键词。

# 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 [
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 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 [
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 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。

分布式的数据库参数支持使用数组定义（通常为了避免多个账号和密码的误解析），例如：

```
return [
    'default'    =>    'mysql',
    'connections'    =>    [
        'mysql'    =>    [
            // 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','abc,def','hello']
            // 数据库连接端口
```

```
            'hostport' => '',
            // 数据库字符集
            'charset'  => 'utf8',
        ],
    ],
];
```

## 读写分离

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` / `readMaster` 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）。

## 主库读取

有些情况下，需要直接从主库读取数据，例如刚写入数据之后，从库数据还没来得及同步完成，你可以使用

```
Db::name('user')
    ->where('id', 1)
    ->update(['name' => 'thinkphp']);
Db::name('user')
    ->master(true)
    ->find(1);
```

不过，实际情况远比这个要复杂，因为你并不清楚后续的方法里面是否还存在相关查询操作，这个时候我们可以配置开启数据库的 `read_master` 配置参数。

```
// 开启自动主库读取
'read_master' => true,
```

开启后，一旦我们对某个数据表进行了写操作，那么当前请求的后续所有对该表的查询都会使用主库读取。

# 查询构造器

# 查询数据

## 查询单个数据

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

如果希望查询数据不存在的时候返回空数组，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)->findOrEmpty();
```

如果希望在没有找到数据后抛出异常可以使用

```
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。

## 查询数据集

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数据集对象，如果需要转换为数组可以使用

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

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。

## 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) {
        //
    }
});
```

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。

## 游标查询

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开销问题。

```
$cursor = Db::table('user')->where('status', 1)->cursor();
foreach($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）。

# 添加数据

## 添加一条数据

可以使用 `save` 方法统一写入数据，自动判断是新增还是更新数据（以写入数据中是否存在主键数据为依据）。

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->save($data);
```

或者使用 `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

> 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：

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->replace()->insert($data);
```

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（通常是二维数组）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：

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
];
Db::name('user')->replace()->insertAll($data);
```

> 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。

```
$data = [
    ['foo' => 'bar', 'bar' => 'foo'],
    ['foo' => 'bar1', 'bar' => 'foo1'],
    ['foo' => 'bar2', 'bar' => 'foo2']
    ...
];
// 分批写入 每次最多100条数据
Db::name('user')
    ->limit(100)
    ->insertAll($data);
```

# 更新数据

## 更新数据

可以使用save方法更新数据

```
Db::name('user')
    ->save(['id' => 1, 'name' => 'thinkphp']);
```

或者使用 `update` 方法。

```
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，则以 `update` 方法为准。

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)
    ->exp('name','UPPER(name)')
    ->update();
```

实际生成的SQL语句：

```
UPDATE `think_user`  SET `name` = UPPER(name)  WHERE  `id` = 1
```

支持使用 `raw` 方法进行数据更新。

```
Db::name('user')
    ->where('id', 1)
    ->update([
        'name'      =>  Db::raw('UPPER(name)'),
        'score'     =>  Db::raw('score-3'),
        'read_time' =>  Db::raw('read_time+1')
    ]);
```

## 自增/自减

可以使用 `inc/dec` 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)
    ->inc('score')
    ->update();

// score 字段加 5
Db::table('think_user')
    ->where('id', 1)
    ->inc('score', 5)
    ->update();

// score 字段减 1
Db::table('think_user')
    ->where('id', 1)
    ->dec('score')
    ->update();

// score 字段减 5
Db::table('think_user')
```

```
    ->where('id', 1)
    ->dec('score', 5)
    ->update();
```

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
```

# 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','查询表达式','查询条件');
```

除了 `where` 方法外，还可以支持 `whereOr` ，用法是一样的。为了更加方便查询，大多数的查询表达式都提供了快捷查询方法。

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 | |
| [NOT] LIKE | 模糊查询 | whereLike/whereNotI |
| [NOT] BETWEEN | （不在）区间查询 | whereBetween/whereN |
| [NOT] IN | （不在）IN 查询 | whereIn/whereNotIn |
| [NOT] NULL | 查询字段是否（不）是NULL | whereNull/whereNotN |
| [NOT] EXISTS | EXISTS查询 | whereExists/whereNo |
| [NOT] REGEXP | 正则（不）匹配查询（仅支持Mysql） | |
| [NOT] BETWEEN TIME | 时间区间比较 | whereBetweenTime |
| > TIME | 大于某个时间 | whereTime |
| < TIME | 小于某个时间 | whereTime |
| >= TIME | 大于等于某个时间 | whereTime |
| <= TIME | 小于等于某个时间 | whereTime |
| EXP | 表达式查询，支持SQL语法 | whereExp |
| find in set | FIND_IN_SET查询 | whereFindInSet |

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：

```
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 'php%')
```

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
```

或者使用快捷查询方法：

```
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)
```

或者使用快捷查询方法：

```
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  AND `nam
e` 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。

推荐使用 `whereExp` 方法查询

```
Db::name('user')->whereExp('id', 'IN (1,3,8) ')->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('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();
```

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 |
| 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 | 用于设置分区信息 | 数组 字符串 |
| replace | 用于设置使用REPLACE方式写入 | 布尔值 |
| extra | 用于设置额外查询规则 | 字符串 |
| duplicate | 用于设置DUPLCATE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

数组方式有两种查询条件类型：关联数组和索引数组。

### 关联数组

主要用于等值 `AND` 条件，例如：

```
// 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
```

### 索引数组

索引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
```

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];
```

> 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')->whereRaw('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，如果需要传入变量，建议配合预处理机制，确保更加安全，例如：

```
Db::table('think_user')
->whereRaw("id=:id and username=:name", ['id' => 1 , '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('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 user
```

可以给某个字段设置别名，例如：

```
Db::table('user')->field('id,nickname as name')->select();
```

执行的SQL语句相当于：

```
SELECT id,nickname as name FROM user
```

### 使用SQL函数

可以在 `fieldRaw` 方法中直接使用函数，例如：

```
Db::table('user')->fieldRaw('id,SUM(score)')->select();
```

执行的SQL相当于：

```
SELECT id,SUM(score) FROM 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('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('user')->field(['id','nickname'=>'name'])->select();
```

执行的SQL相当于：

```
SELECT id,nickname as name FROM user
```

## 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('user')->select();
Db::table('user')->field('*')->select();
```

上面的用法是等效的，都相当于执行SQL：

```
SELECT * FROM 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('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('user')->withoutField('content')->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('user')->withoutField('user_id,content')->select();
//或者用
Db::table('user')->withoutField(['user_id','content'])->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('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('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('user')
->where('score',100)
->limit(3)
->update(['level'=>'A']);
```

如果用于 `insertAll` 方法的话，则可以分批多次写入，每次最多写入 `limit` 方法指定的数量。

```
Db::table('user')
->limit(100)
->insertAll($userList);
```

## 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('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('article')->limit(0,10)->select();
// 查询第二页数据
Db::table('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('article')->page(1,10)->select();
// 查询第二页数据
Db::table('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('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('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('user')
->where('status', 1)
->order('id', 'desc')
->limit(5)
->select();
```

```
 SELECT * FROM `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('user')
->where('status', 1)
->order(['order','id'=>'desc'])
->limit(5)
->select();
```

最终的查询SQL可能是

```
SELECT * FROM `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('user')
->where('status', 1)
->order('id', 'desc')
->limit(5)
->delete();
```

生成的SQL

```
DELETE FROM `user` WHERE `status` = 1 ORDER BY `id` desc LIMIT 5
```

如果你需要在 `order` 方法中使用mysql函数的话，必须使用下面的方式：

```
Db::table('user')
->where('status', 1)
->orderRaw("field(name,'thinkphp','onethink','kancloud')")
->limit(5)
->select();
```

# 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('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 score GROUP BY user_id
```

也支持对多个字段进行分组，例如：

```
Db::table('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 user GROUP BY user_id,test_ti
me
```

# 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('score')
    ->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 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，只能是字符串。

### 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_user_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('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('user')->cache(true,60)->find();
// 或者使用下面的方式 是等效的
Db::table('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('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('user')->cache('key',60)->find();
$data = \think\facade\Cache::get('key');
```

`cache` 方法支持设置缓存标签，例如：

```
Db::table('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('user')->cache('user_data')->select([1,3,5]);
Db::table('user')->cache('user_data')->update(['id'=>1,'name'=>'thinkphp']);
Db::table('user')->cache('user_data')->select([1,3,5]);
```

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。

如果使用主键进行查询和更新(或者删除）的话，无需指定缓存标识会自动更新缓存

```
Db::table('user')->cache(true)->find(1);
Db::table('user')->cache(true)->where('id', 1)->update(['name'=>'thinkphp']);
Db::table('user')->cache(true)->find(1);
```

# 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('user')->fetchSql(true)->find(1);
```

输出结果为：

```
SELECT * FROM user where `id` = 1
```

> 对于某些NoSQL数据库可能不支持fetchSql方法

# force

force 方法用于数据集的强制索引操作，例如：

```
Db::table('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 `MySQL` 数据库的分区查询，用法如下：

```
// 用于查询
Db::name('log')
    ->partition(['p1','p2'])
    ->select();

// 用于写入
Db::name('user')
    ->partition('p1')
    ->insert(['name' => 'think', 'score' => 100']);
```

# failException

`failException` 设置查询数据为空时是否需要抛出异常，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']);
```

# replace

`replace` 方法用于设置 `MySQL` 数据库 `insert` 方法或者 `insertAll` 方法写入数据的时候是否适用 `REPLACE` 方式。

```
Db::name('user')
    ->replace()
    ->insert($data);
```

# extra

`extra` 方法可以用于 `CURD` 查询，例如：

```
Db::name('user')
    ->extra('IGNORE')
    ->insert(['name' => 'think']);

Db::name('user')
    ->extra('DELAYED')
    ->insert(['name' => 'think']);

Db::name('user')
    ->extra('SQL_BUFFER_RESULT')
    ->select();
```

# duplicate

用于设置 `DUPLICATE` 查询，用法示例：

```
Db::name('user')
    ->duplicate(['score' => 10])
    ->insert(['name' => 'think']);
```

# procedure

`procedure` 方法用于设置当前查询是否为存储过程查询，用法如下：

```
$resultSet = Db::procedure(true)
    ->query('call procedure_name');
```

# 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

## 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：

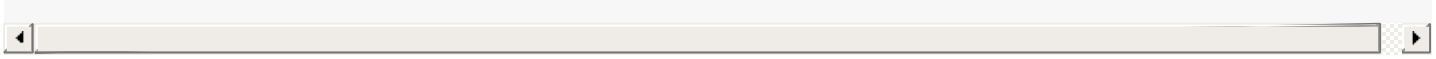
```
Db::table('think_user')->where('id',10)->sum('score');
```

生成的SQL语句是：

```
SELECT SUM(score) AS tp_sum FROM `think_user` LIMIT 1
```

如果你要使用 `group` 进行聚合查询，需要自己实现查询，例如：

```
Db::table('score')->field('user_id,SUM(score) AS sum_score')->group('user_id')->
select();
```



# 分页查询

## 分页实现

`ThinkPHP` 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)->order('id', 'desc')->paginate(10);

// 渲染模板输出
return view('index', ['list' => $list]);
```

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 = Db::name('user')->where('status',1)->order('id', 'desc')->paginate(10);

// 获取分页显示
$page = $list->render();

return view('index', ['list' => $list, 'page' => $page]);
```

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 = Db::name('user')->where('status',1)->order('id' ,'desc')->paginate(10);
// 获取总记录数
$count = $list->total();
return view('index', ['list' => $list, 'count' => $count]);
```

## 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 = Db::name('user')->where('status',1)->paginate(10,1000);
// 获取分页显示
$page = $list->render();

return view('index', ['list' => $list, 'page' => $page]);
```

> 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。

```
$list = Db::name('user')->where('status',1)->order('id', 'desc')->paginate()->each(function($item, $key){
    $item['nickname'] = 'think';
    return $item;
});
```

如果是模型类操作分页数据的话， `each` 方法的闭包函数中不需要使用返回值，例如：

```
$list = User::where('status',1)->order('id', 'desc')->paginate()->each(function($item, $key){
    $item->nickname = 'think';
});
```

## 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 = Db::name('user')->where('status',1)->order('id', 'desc')->paginate(10, true);

// 渲染模板输出
return view('index', ['list' => $list]);
```

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 |

分页参数的设置可以在调用分页方法的时候传入，例如：

```
$list = Db::name('user')->where('status',1)->paginate([
    'list_rows'=> 20,
    '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

## 大数据分页

对于大量数据的分页查询，系统提供了一个高性能的 `paginateX` 分页查询方法，用法和 `paginate` 分页查询存在一定区别。如果你要分页查询的数据量在百万级以上，使用 `paginateX` 方法会有明显的提升，尤其是在分页数较大的情况下。并且由于针对大数据量而设计，该分页查询只能采用简洁分页模式，所以没有总数。

> 分页查询的排序字段一定要使用索引字段，并且是连续的整型，否则会有数据遗漏。

主要场景是针对主键进行分页查询，默认使用主键倒序查询分页数据。

```
$list = Db::name('user')->where('status',1)->paginateX(20);
```

也可以在查询的时候可以指定主键和排序

```
$list = Db::name('user')->where('status',1)->paginateX(20, 'id', 'desc');
```

查询方法会执行两次查询，第一次查询用于查找满足当前查询条件的最大或者最小值，然后配合主键查询条件来进行分页数据查询。

## 自定义分页类

如果你需要自定义分页，可以扩展一个分页驱动。
然后在 `provider.php` 定义文件中重新绑定

```
return [
    'think\Paginator'    =>    'app\common\Bootstrap'
];
```

# 时间查询

## 时间比较

> 框架内置了常用的时间查询方法，并且可以自动识别时间字段的类型，所以无论采用什么类型的时间字段，都可以统一使用本章的时间查询用法。

### 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();
```

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();
```

## 查询某个时间区间

针对时间的区间查询，系统还提供了 `whereBetweenTime/whereNotBetweenTime` 快捷方法。

```
// 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();
```

```
// 查询不是2017年上半年注册的用户
Db::name('user')
    ->whereNotBetweenTime('create_time', '2017-01-01', '2017-06-30')
    ->select();
```

## 查询某年

查询今年注册的用户

```
Db::name('user')
    ->whereYear('create_time')
    ->select();
```

查询去年注册的用户

```
Db::name('user')
    ->whereYear('create_time', 'last year')
    ->select();
```

查询某一年的数据使用

```
// 查询2018年注册的用户
Db::name('user')
    ->whereYear('create_time', '2018')
    ->select();
```

## 查询某月

查询本月注册的用户

```
Db::name('user')
    ->whereMonth('create_time')
    ->select();
```

查询上月注册用户

```
Db::name('user')
    ->whereMonth('create_time','last month')
    ->select();
```

查询2018年6月注册的用户

```
Db::name('user')
    ->whereMonth('create_time', '2018-06')
    ->select();
```

## 查询某周

查询本周数据

```
Db::name('user')
    ->whereWeek('create_time')
    ->select();
```

查询上周数据

```
Db::name('user')
    ->whereWeek('create_time', 'last week')
    ->select();
```

查询指定某天开始的一周数据

```
// 查询2019-1-1到2019-1-7的注册用户
Db::name('user')
    ->whereWeek('create_time', '2019-1-1')
    ->select();
```

## 查询某天

查询当天注册的用户

```
Db::name('user')
    ->whereDay('create_time')
    ->select();
```

查询昨天注册的用户

```
Db::name('user')
    ->whereDay('create_time', 'yesterday')
    ->select();
```

查询某天的数据使用

```
// 查询2018年6月1日注册的用户
Db::name('user')
    ->whereDay('create_time', '2018-06-01')
    ->select();
```

## 时间字段区间比较

可以支持对两个时间字段的区间比较

```
// 查询有效期内的活动
Db::name('event')
    ->whereBetweenTimeField('start_time', 'end_time')
    ->select();
```

上面的查询相当于

```
// 查询有效期内的活动
Db::name('event')
    ->whereTime('start_time', '<=', time())
    ->whereTime('end_time', '>=', time())
    ->select();
```

## 自定义时间查询规则

你可以通过在数据库配置文件中设置 `time_query_rule` 添加自定义的时间查询规则，

```
'time_query_rule'    =>    [
    'hour'    =>    ['1 hour ago', 'now'],
],
```

# 高级查询

## 快捷查询

快捷查询方式是一种多字段相同查询条件的简化写法，可以进一步简化查询条件的写法，在多个字段之间用 `|` 分割表示 `OR` 查询，用 `&` 分割表示 `AND` 查询，可以实现下面的查询，例如：

```
Db::table('think_user')
    ->where('name|title','like','thinkphp%')
    ->where('create_time&update_time','>',0)
    ->find();
```

生成的查询SQL是：

```
SELECT * FROM `think_user` WHERE ( `name` LIKE 'thinkphp%' OR `title` LIKE 'thi
nkphp%' ) AND ( `create_time` > 0 AND `update_time` > 0 ) LIMIT 1
```

> 快捷查询支持所有的查询表达式。

## 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 '%thi
nkphp' AND `id` > 0 AND `status` = '1'
```

数组方式如果使用 `exp` 查询的话，一定要用 `raw` 方法。

```
Db::table('think_user')
    ->where([
```

```
        ['name', 'like', 'thinkphp%'],
        ['title', 'like', '%thinkphp'],
        ['id', 'exp', Db::raw('>score')],
        ['status', '=', 1],
    ])
    ->select();
```

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` LIKE '%t
hinkphp' 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` LIKE '%t
hinkphp' ) 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%')
    ->where(function ($query) {
        $query->where('id', '<', 10);
    })
    ->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE  `name` LIKE 'thinkphp%' AND ( `id` < 10 )
```

## 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')
    ->whereRaw('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')
    ->whereRaw('id > :id AND name LIKE :name ', ['id' => 0, 'name' => 'thinkphp%'])
    ->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();
```

支持数组方式比较多个字段

```
Db::name('user')->whereColumn([
    ['title', '=', 'name'],
    ['update_time', '>=', 'create_time'],
])->select();
```

生成的SQL语句是：

```
SELECT * FROM `think_user` WHERE ( `name` = `nickname` AND `update_time` > `cre
ate_time` )
```

## 动态查询

查询构造器还提供了动态查询机制，用于简化查询条件，包括：

| 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

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,Score.sco
re 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,Score.sco
re 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,Profi
le.email,Score.score FROM think_user User INNER JOIN think_profile Profile ON P
rofile.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.phone,P
rofile.email,Score.score FROM think_user member INNER JOIN think_profile Profil
e ON Profile.user_id=member.id INNER JOIN think_socre Score ON Score.user_id=Pr
ofile.id WHERE Score.score > 80
```

# JSON字段

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);
```

由于JSON字段的属性类型并不会自动获取，所以，如果是整型数据查询的话，可以设置JSON字段类型，例如：

```
$user = Db::name('user')
    ->json(['info'])
    ->where('info->user_id', 10)
    ->setFieldType(['info->user_id' => 'int'])
    ->find();
dump($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_profile` WHE
RE `status` = 1 )
```

```
Db::table('think_user')
    ->whereExists(function ($query) {
        $query->table('think_profile')->where('status', 1);
    })->find();
```

生成的SQL语句为

```
SELECT * FROM `think_user` WHERE EXISTS ( SELECT * FROM `think_profile` WHERE `
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，返回查询结果数据集（数组）。

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` 方法默认是在读服务器执行，而不管你的SQL语句是什么。

如果希望从主库读取，可以使用

```
Db::master(true)->query("select * from think_user where status=1");
```

新版的 `query` 方法可以支持的链式操作方法包括：

| 链式方法 | 说明 |
| --- | --- |
| cache | 查询缓存 |
| master | 是否主库查询（分布式有效） |
| procedure | 存储过程查询 |

## `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，而不管你的SQL语句是什么。

## 参数绑定

支持在原生查询的时候使用参数绑定，包括问号占位符或者命名占位符，例如：

```
Db::query("select * from think_user where id=? AND status=?", [8, 1]);
// 命名绑定
Db::execute("update think_user set name=:name where status=:status", ['name' =>
'thinkphp', 'status' => 1]);
```

> 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 |

使用下面的方法注册数据库查询事件

```
\think\facade\Db::event('before_select', function ($query) {
    // 事件处理
    return $result;
});
```

同一个查询事件可以注册多个响应执行。查询事件在新版里面也已经被事件系统接管了，因此如果你注册了一个 `before_select` 查询事件监听，底层其实是向标识为 `db.before_select` 的事件注册了一个监听。

> 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。

# 获取器

## 获取器

Db类也可以支持获取器定义，例如：

```
Db::name('user')->withAttr('name', function($value, $data) {
    return strtolower($value);
})->select();
```

获取器方法支持传入两个参数，第一个参数是当前字段的值，第二个参数是所有的数据。

上面的代码，查询的数据集数据中的 `name` 字段的值会统一进行小写转换。

> `withAttr` 方法可以多次调用，对多个字段定义获取器。

支持对 `JSON` 字段定义获取器，例如：

```
$user = Db::name('user')
    ->json(['info'])
    ->withAttr('info.name', function($value, $data) {
        return strtolower($value);
    })->find(1);
dump($user);
```

查询结果返回的时候，会自动对 `info` 字段（ `JSON` 字段）的 `name` 属性使用获取器操作。

# 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。

可以支持MySQL的 `XA` 事务用于实现全局（分布式）事务，你可以使用：

```
Db::transactionXa(function () {
    Db::connect('db1')->table('think_user')->delete(1);
    Db::connect('db2')->table('think_user')->delete(1);
}, [Db::connect('db1'),Db::connect('db2')]);
```

> 要确保你的数据表引擎为 `InnoDB` ，并且开启XA事务支持。

# 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) {

}
```

系统会自动判断当前查询是否为存储过程调用，但可以使用 `procedure` 方法明确指定当前查询为存储过程查询，提高查询效率。

```
$resultSet = Db::procedure(true)
    ->query('call procedure_name');
```

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_param)',
[
    '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

数据库的查询结果默认返回数据集对象。

```
// 获取数据集
$users = Db::name('user')->select();
// 遍历数据集
foreach($users as $user){
    echo $user['name'];
    echo $user['id'];
}
```

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（继承 `think\Collection` ），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 |
| push | 在结尾插入一个元素 |
| 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 |
| order | 指定字段排序 |
| shuffle | 将数据打乱 |
| slice | 截取数据中的一部分 |
| map | 用回调函数处理数组中的元素 |
| where | 根据字段条件过滤数组中的元素 |
| whereLike | Like查询过滤元素 |
| whereNotLike | Not Like过滤元素 |
| whereIn | IN查询过滤数组中的元素 |
| whereNotIn | Not IN查询过滤数组中的元素 |
| whereBetween | Between查询过滤数组中的元素 |
| whereNotBetween | Not Between查询过滤数组中的元素 |

# 数据库驱动

## 数据库驱动

如果你需要自定义数据库驱动，需要自定义实现 `Connection` 类（或者继承 `think\db\Connection` ）和 `Builder` 类（或者继承 `think\db\Builder` ），对于特殊的驱动，可能还需要实现 `Query` 类（或者继承 `think\db\Query` ）。

> 具体数据库驱动的实现，要根据你的自定义 `Connection` 类来决定。可以参考官方的 `oracle` 驱动和 `mongo` 驱动的实现。

一旦自定义了数据库驱动，例如你自定义实现了 `think\mongo\Connection` 你需要在数据库配置文件中配置：

```
'type'  =>   'think\mongo\Connection',
'query' =>   'think\mongo\Query',
```

# 模型

# 定义

## 模型定义

定义一个模型类很简单，例如下面是一个 `User` 模型：

```
<?php
namespace app\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。

如果你希望给模型类添加后缀，需要设置 `name` 属性或者 `table` 属性。

```
<?php
namespace app\model;

use think\Model;

class UserModel extends Model
{
    protected $name = 'user';
}
```

## 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\model;

use think\Model;

class User extends Model
{
    protected $pk = 'uid';
}
```

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\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` 属性使用用配置参数名（需要在数据库配置文件中的 `connections` 参数中添加对应标识）。

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（相当于不带数据表前后缀的表名，默认为当前模型类名） |
| table | 数据表名（默认自动获取） |
| suffix | 数据表后缀（默认为空） |
| 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 | 模型允许写入的字段列表（数组） |
| schema | 模型对应数据表字段及类型 |
| type | 模型需要自动转换的字段及类型 |
| strict | 是否严格区分字段大小写（默认为true） |
| disuse | 数据表废弃字段（数组） |

> 模型不支持对数据表的前缀单独设置，并且也不推荐使用数据表的前缀设计，应该用不同的库区分。当你的数据表没有前缀的时候， `name` 和 `table` 属性的定义是没有区别的，定义任何一个即可。

## 模型初始化

模型支持初始化，只需要定义 `init` 方法，例如：

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 模型初始化
    protected static function init()
    {
        //TODO:初始化内容
    }
}
```

> `init` 必须是静态方法，并且只在第一次实例化的时候执行，并且只会执行一次

## 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，数据库的所有查询构造器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。

模型的操作方法无需和数据库查询一样调用必须首先调用 `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、自动时间写入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。

## 动态切换后缀

新版模型增加了一个数据表后缀属性，可以用于多语言或者数据分表的模型查询，省去为多个相同结构的表定义多个模型的麻烦。

默认的数据表后缀可以在模型类里面直接定义 `suffix` 属性。

```
<?php
namespace app\model;

use think\Model;

class Blog extends Model
{
    // 定义默认的表后缀（默认查询中文数据）
    protected $suffix = _cn';
}
```

> 你在模型里面定义的 `name` 和 `table` 属性无需包含后缀定义

模型提供了动态切换方法 `suffix` 和 `setSuffix` ，例如：

```
// suffix方法用于静态查询
$blog = Blog::suffix('_en')->find();
$blog->name = 'test';
$blog->save();

// setSuffix用于动态设置
$blog = new Blog($data);
$blog->setSuffix('_en')->save();
```

## 模型方法依赖注入

如果你需要对模型的方法支持依赖注入，可以把模型的方法改成闭包的方式，例如，你需要对获取器方法增加依赖注入

```
public function getTestFieldAttr($value,$data) {
    return $this->invoke(function(Request $request)  use($value,$data) {
        return $data['name'] . $request->action();
    });
}
```

不仅仅是获取器方法，在任何需要依赖注入的方法都可以改造为调用 `invoke` 方法的方式， `invoke` 方法第二个参数用于传入需要调用的（数组）参数。

如果你需要直接调用某个已经定义的模型方法（假设已经使用了依赖注入），可以使用

```
protected function bar($name, Request $request) {
    // ...
}

protected function invokeCall(){
    return $this->invoke('bar',['think']);
}
```

# 模型字段

## 模型字段

模型的数据字段和对应数据表的字段是对应的，默认会自动获取（包括字段类型），但自动获取会导致增加一次查询，因此你可以在模型中明确定义字段信息避免多一次查询的开销。

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 设置字段信息
    protected $schema = [
        'id'          => 'int',
        'name'        => 'string',
        'status'      => 'int',
        'score'       => 'float',
        'create_time' => 'datetime',
        'update_time' => 'datetime',
    ];
}
```

> 字段类型的定义可以使用PHP类型或者数据库的字段类型都可以，字段类型定义的作用主要用于查询的参数自动绑定类型。

> 时间字段尽量采用实际的数据库类型定义，便于时间查询的字段自动识别。如果是 `json` 类型直接定义为 `json` 即可。

如果你没有定义 `schema` 属性的话，可以在部署完成后运行如下指令。

```
php think optimize:schema
```

运行后会自动生成数据表的字段信息缓存。使用命令行缓存的优势是Db类的查询仍然有效。

## 字段类型

`schema` 属性一旦定义，就必须定义完整的数据表字段类型。
如果你只希望对某个字段定义需要自动转换的类型，可以使用 `type` 属性，例如：

```php
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 设置字段自动转换类型
    protected $type = [
        'score'       => 'float',
    ];
}
```

`type` 属性定义的不一定是实际的字段，也有可能是你的字段别名。

## 废弃字段

如果因为历史遗留问题 ，你的数据表存在很多的废弃字段，你可以在模型里面定义这些不再使用的字段。

```php
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 设置废弃字段
    protected $disuse = [ 'status', 'type' ];
}
```

在查询和写入的时候会忽略定义的 `status` 和 `type` 废弃字段。

## 获取数据

在模型外部获取数据的方法如下

```php
$user = User::find(1);
echo $user->create_time;
echo $user->name;
```

由于模型类实现了 `ArrayAccess` 接口，所以可以当成数组使用。

```php
$user = User::find(1);
echo $user['create_time'];
echo $user['name'];
```

如果你是在模型内部获取数据的话，需要改成：

```
$user = $this->find(1);
echo $user->getAttr('create_time');
echo $user->getAttr('name');
```

否则可能会出现意想不到的错误。

## 模型赋值

可以使用下面的代码给模型对象赋值

```
$user = new User();
$user->name = 'thinkphp';
$user->score = 100;
```

该方式赋值会自动执行模型的修改器，如果不希望执行修改器操作，可以使用

```
$data['name'] = 'thinkphp';
$data['score'] = 100;
$user = new User($data);
```

或者使用

```
$user = new User();
$data['name'] = 'thinkphp';
$data['score'] = 100;
$user->data($data);
```

`data` 方法支持使用修改器

```
$user = new User();
$data['name'] = 'thinkphp';
$data['score'] = 100;
$user->data($data, true);
```

如果需要对数据进行过滤，可以使用

```
$user = new User();
$data['name'] = 'thinkphp';
$data['score'] = 100;
```

```
$user->data($data, true, ['name','score']);
```

表示只设置 `data` 数组的 `name` 和 `score` 数据。

## 严格区分字段大小写

默认情况下，你的模型数据名称和数据表字段应该保持严格一致，也就是说区分大小写。

```
$user = User::find(1);
echo $user->create_time;  // 正确
echo $user->createTime;  // 错误
```

> 严格区分字段大小写的情况下，如果你的数据表字段是大写，模型获取的时候也必须使用大写。

如果你希望在获取模型数据的时候不区分大小写（前提是数据表的字段命名必须规范，即小写+下划线），可以设置模型的 `strict` 属性。

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 模型数据不区分大小写
    protected $strict = false,
}
```

你现在可以使用

```
$user = User::find(1);
// 下面两种方式都有效
echo $user->createTime;
echo $user->create_time;
```

# 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` 方法成功会返回 `true` ，并且只有当 `before_insert` 事件返回 `false` 的时候返回 `false` ，一旦有错误就会抛出异常。所以无需判断返回类型。

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'
]);
```

默认只会写入数据表已有的字段，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。

## `Replace` 写入

`save` 方法可以支持 `replace` 写入。

```
$user           = new User;
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->replace()->save();
```

## 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。

## 批量增加数据

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。

## 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` 方法默认会过滤不是数据表的字段信息，可以在第二个参数可以传入允许写入的字段列表，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
```

支持 `replace` 操作，使用下面的方法：

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

## 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::find(1);
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->save();
```

> `save` 方法成功返回 `true` ，并只有当 `before_update` 事件返回 `false` 的时候返回 `false` ，有错误则会抛出异常。

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();
```

`save` 方法更新数据，只会更新变化的数据，对于没有变化的数据是不会进行重新更新的。如果你需要强制更新数据，可以使用下面的方法：

```
$user = User::find(1);
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->force()->save();
```

这样无论你的修改后的数据是否和之前一样都会强制更新该字段的值。

如果要执行SQL函数更新，可以使用下面的方法

```
$user = User::find(1);
$user->name     = 'thinkphp';
$user->email    = 'thinkphp@qq.com';
$user->score    =  Db::raw('score+1');
```

```
$user->save();
```

## 字段过滤

默认情况下会过滤非数据表字段的数据，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 = User::find(1);
// post数组中只有name和email字段会写入
$user->allowField(['name', 'email'])->save($_POST);
```

最佳用法是在传入模型数据之前就进行过滤，例如：

```
$user = User::find(1);
// post数组中只有name和email字段会写入
$data = Request::only(['name','email']);
$user->save($data);
```

## 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。

## 直接更新（静态方法）

使用模型的静态 `update` 方法更新：

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

> 模型的 `update` 方法返回模型的对象实例

如果你的第一个参数中包含主键数据，可以无需传入第二个参数（更新条件）

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

如果你需要只允许更新指定字段，可以使用

```
User::update(['name' => 'thinkphp', 'email' => 'thinkphp@qq.com'], ['id' => 1],
['name']);
```

上面的代码只会更新 `name` 字段的数据。

## 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；

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。

> 不要调用 `save` 方法进行多次数据写入。

## 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或者不查询直接更新，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::find(1);
$user->delete();
```

> `delete` 方法返回布尔值

## 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]);
```

> 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。但如果直接调用查询对象的方法，IDE可能无法完成自动提示。

## 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::find(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;
```

模型使用 `find` 方法查询，如果数据不存在返回 `Null` ，否则返回当前模型的对象实例。如果希望查询数据不存在则返回一个空模型，可以使用。

```
$user = User::findOrEmpty(1);
```

你可以用 `isEmpty` 方法来判断当前是否为一个空模型。

```
$user = User::where('name', 'thinkphp')->findOrEmpty();
if (!$user->isEmpty()) {
    echo $user->name;
}
```

> 如果你是在模型内部获取数据，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
```

```
$list = User::select([1,2,3]);
// 对数据集进行遍历操作
foreach($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 `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');
```

注意，如果你的字段不是数字类型，是使用 `max` / `min` 的时候，需要加上第二个参数。

```
User::max('name', false);
```

## 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。

## 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。

# 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\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\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

支持在模型里面设置 `globalScope` 属性，定义全局的查询范围

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 定义全局的查询范围
    protected $globalScope = ['status'];

    public function scopeStatus($query)
    {
        $query->where('status',1);
    }
}
```

然后，执行下面的代码：

```
$user = User::find(1);
```

最终的查询条件会是

```
status = 1 AND id = 1
```

如果需要动态关闭所有的全局查询范围，可以使用：

```
// 关闭全局查询范围
User::withoutGlobalScope()->select();
```

可以使用 `withoutGlobalScope` 方法动态关闭部分全局查询范围。

```
User::withoutGlobalScope(['status'])->select();
```

# JSON字段

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\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::find(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
```

如果你需要查询的JSON属性是整型类型的话，可以在模型类里面定义JSON字段的属性类型，就会自动进行相应类型的参数绑定查询。

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 设置json类型字段
    protected $json = ['info'];

    // 设置JSON字段的类型
    protected $jsonType = [
        'info->user_id' =>  'int'
    ];
}
```

没有定义类型的属性默认为字符串类型，因此字符串类型的属性可以无需定义。

可以设置模型的 `JSON` 数据返回数组，只需要在模型设置 `jsonAssoc` 属性为 `true` 。

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    // 设置json类型字段
    protected $json = ['info'];
```

```
    // 设置JSON数据返回数组
    protected $jsonAssoc = true;
}
```

设置后，查询代码调整为：

```
$user = User::find(1);
echo $user->name; // thinkphp
echo $user->info['email']; // thinkphp@qq.com
echo $user->info['nickname']; // 流年
```

## 更新JSON数据

```
$user = User::find(1);
$user->name = 'kancloud';
$user->info->email = 'kancloud@qq.com';
$user->info->nickname = 'kancloud';
$user->save();
```

如果设置模型的 `JSON` 数据返回数组，那么更新操作需要调整如下。

```
$user = User::find(1);
$user->name = 'kancloud';
$info['email'] = 'kancloud@qq.com';
$info['nickname'] = 'kancloud';
$user->info = $info;
$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（该方法必须为 `public` 类型），方法命名规范为：

```
get FieldName Attr
```

`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
namespace app\model;

use think\Model;

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::find(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
namespace app\model;

use think\Model;

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::find(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::find(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());
```

## 动态获取器

可以支持对模型使用动态获取器，无需在模型类中定义获取器方法。

```
User::withAttr('name', function($value, $data) {
    return strtolower($value);
})->select();
```

> `withAttr` 方法支持多次调用，定义多个字段的获取器。另外注意， `withAttr` 方法之后不能再使用模型的查询方法，必须使用Db类的查询方法。

> 如果同时还在模型里面定义了相同字段的获取器，则动态获取器优先，也就是可以临时覆盖定义某个字段的获取器。

支持对关联模型的字段使用动态获取器，例如：

```
User::with('profile')->withAttr('profile.name', function($value, $data) {
    return strtolower($value);
})->select();
```

> 注意：对于 `MorphTo` 关联不支持使用动态获取器。

并且支持对 `JSON` 字段使用获取器，例如在模型中定义了 `JSON` 字段的话：

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

use think\Model;

class User extends Model
{
    // 设置json类型字段
    protected $json = ['info'];
}
```

可以使用下面的代码定义JSON字段的获取器。

```
User::withAttr('info.name', function($value, $data) {
    return strtolower($value);
})->select();
```

可以在查询之后使用 `withAttr` 方法，例如：

```
User::select()->withAttr('name', function($value, $data) {
    return strtolower($value);
});
```

# 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：

```
set FieldName Attr
```

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；

例如：

```php
<?php
namespace app\model;

use think\Model;

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

```php
$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：

```
namespace app\model;

use think\Model;

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。

如果你需要在修改器中修改其它数据，可以使用

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    public function setTestFieldAttr($value, $data)
    {
        $this->set('other_field', $data['some_field']);
    }
}
```

上面的例子，在 `test_field` 字段的修改器中修改了 `other_field` 字段数据，并且没有返回值（表示不对 `test_field` 字段做任何修改）。

## 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);
```

# 搜索器

## 搜索器

搜索器的作用是用于封装字段（或者搜索标识）的查询条件表达式，一个搜索器对应一个特殊的方法（该方法必须是 `public` 类型），方法命名规范为：

```
search FieldName Attr
```

`FieldName` 为数据表字段的驼峰转换，搜索器仅在调用 `withSearch` 方法的时候触发。
搜索器的场景包括：

- 限制和规范表单的搜索条件；
- 预定义查询条件简化查询；

例如，我们需要给User模型定义 `name` 字段和时间字段的搜索器，可以使用：

```php
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    public function searchNameAttr($query, $value, $data)
    {
        $query->where('name','like', $value . '%');
    }

    public function searchCreateTimeAttr($query, $value, $data)
    {
        $query->whereBetweenTime('create_time', $value[0], $value[1]);
    }
}
```

然后，我们可以使用下面的查询

```php
User::withSearch(['name','create_time'], [
        'name'          =>  'think',
        'create_time'   =>  ['2018-8-1','2018-8-5'],
        'status'        =>  1
    ])
    ->select();
```

最终生成的SQL语句类似于

```
SELECT * FROM `think_user` WHERE  `name` LIKE 'think%' AND `create_time` BETWEE
N '2018-08-01 00:00:00' AND '2018-08-05 00:00:00'
```

可以看到查询条件中并没有 `status` 字段的数据，因此可以很好的避免表单的非法查询条件传入，在这个示例中仅能使用 `name` 和 `create_time` 条件进行查询。

> 事实上，除了在搜索器中使用查询表达式外，还可以使用其它的任何查询构造器以及链式操作。

例如，你需要通过表单定义的排序字段进行搜索结果的排序，可以使用

```php
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    public function searchNameAttr($query, $value, $data)
    {
        $query->where('name','like', $value . '%');
        if (isset($data['sort'])) {
            $query->order($data['sort']);
        }
    }

    public function searchCreateTimeAttr($query, $value, $data)
    {
        $query->whereBetweenTime('create_time', $value[0], $value[1]);
    }
}
```

然后，我们可以使用下面的查询

```php
User::withSearch(['name','create_time', 'status'], [
        'name'          =>  'think',
        'create_time'   =>  ['2018-8-1','2018-8-5'],
        'status'        =>  1,
        'sort'          =>  ['status'=>'desc'],
    ])
    ->select();
```

最终查询的SQL可能是

```
SELECT * FROM `think_user` WHERE  `name` LIKE 'think%' AND `create_time` BETWEE
N '2018-08-01 00:00:00' AND '2018-08-05 00:00:00' ORDER BY `status` DESC
```

你可以给搜索器定义字段别名，例如：

```
User::withSearch(['name'=>'nickname','create_time', 'status'], [
        'nickname'      =>  'think',
        'create_time'   =>  ['2018-8-1','2018-8-5'],
        'status'        =>  1,
        'sort'          =>  ['status'=>'desc'],
    ])
    ->select();
```

> 搜索器通常会和查询范围进行比较，搜索器无论定义了多少，只需要一次调用，查询范围如果需要组合查询的时候就需要多次调用。

# 数据集

## 数据集

> 模型的 `select` 查询方法返回数据集对象 `think\model\Collection` ，该对象继承自 `think\Collection` ，因此具有数据库的数据集类的所有方法，而且还提供了额外的模型操作方法。

基本用法和数组一样，例如可以遍历和直接获取某个元素。

```
// 模型查询返回数据集对象
$list = User::where('id', '>', 0)->select();
// 获取数据集的数量
echo count($list);
// 直接获取其中的某个元素
dump($list[0]);
// 遍历数据集对象
foreach ($list as $user) {
    dump($user);
}
// 删除某个元素
unset($list[0]);
```

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：

```
$users = User::select();
if($users->isEmpty()){
    echo '数据集为空';
}
```

你可以使用模型的 `hidden` / `visible` / `append` / `withAttr` 方法进行数据集的输出处理，例如：

```
// 模型查询返回数据集对象
$list = User::where('id', '>', 0)->select();
// 对输出字段进行处理
$list->hidden(['password'])
    ->append(['status_text'])
    ->withAttr('name', function($value, $data) {
        return strtolower($value);
    });
dump($list);
```

如果需要对数据集的结果进行筛选，可以使用：

```
// 模型查询返回数据集对象
$list = User::where('id', '>', 0)->select()
    ->where('name', 'think')
    ->where('score', '>', 80);
dump($list);
```

支持 `whereLike` / `whereIn` / `whereBetween` 等快捷方法。

```
// 模型查询返回数据集对象
$list = User::where('id', '>', 0)->select()
    ->whereLike('name', 'think%')
    ->whereBetween('score', [80,100]);
dump($list);
```

支持数据集的 `order` 排序操作。

```
// 模型查询返回数据集对象
$list = User::where('id', '>', 0)->select()
    ->where('name', 'think')
    ->where('score', '>', 80)
    ->order('create_time','desc');
dump($list);
```

支持数据集的 `diff/intersect` 操作。

```
// 模型查询返回数据集对象
$list1 = User::where('status', 1)->field('id,name')->select();
$list2 = User::where('name', 'like', 'think')->field('id,name')->select();
// 计算差集
dump($list1->diff($list2));
// 计算交集
dump($list1->intersect($list2));
```

## 批量删除和更新数据

支持对数据集的数据进行批量删除和更新操作，例如：

```
$list = User::where('status', 1)->select();
$list->update(['name' => 'php']);

$list = User::where('status', 1)->select();
$list->delete();
```

# 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\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\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\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\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：

```
namespace app\model;

use think\Model;

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::find(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\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::find(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。

支持动态设置只读字段，例如：

```
$user = User::find(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\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。
可以支持 `defaultSoftDelete` 属性来定义软删除字段的默认值，在此之前的版本，软删除字段的默认值必须为 `null` 。

```
<?php
namespace app\model;

use think\Model;
use think\model\concern\SoftDelete;

class User extends Model
{
    use SoftDelete;
    protected $deleteTime = 'delete_time';
    protected $defaultSoftDelete = 0;
}
```

> 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::find(1);
// 软删除
$user->delete();
// 真实删除
$user->force()->delete();
```

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();
```

> 软删除的删除操作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。

```
$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
namespace app\model;

use think\Model;

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
namespace app\model;

use think\Model;

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
namespace app\model;
```

```
use think\Model;

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。

# 模型输出

## 模型输出

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\controller;

use app\model\User;
use think\facade\View;

class Index
{
    public function index()
    {
        $user = User::find(1);
        View::assign('user', $user);

        return View::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::select();
$list = $list->toArray();
```

可以在查询之前定义 `hidden` / `visible` / `append` 方法，例如：

```
dump(User::where('id',10)->hidden(['create_time','update_time'])->append(['stat
us_text'])->find()->toArray());
```

注意，必须要首先调用一次Db类的方法后才能调用 `hidden` / `visible` / `append` 方法。

## 追加关联属性

支持追加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：

```
$user = User::with('profile')->find(1);
// 隐藏profile关联属性的email属性
dump($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::find(1);
echo $user->toJson();
```

可以设置需要隐藏的字段，例如：

```
$user = User::find(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::find(1);
echo $user->append(['status_text'])->toJson();
```

设置允许输出的属性：

```
$user = User::find(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::find(1));
```

输出结果类似于：

```
{"id":"1","name":"","title":"","status":"1","update_time":"1430409600","score":
"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::find(1);
```

输出的结果和上面是一样的。

# 模型事件

## 模型事件

模型事件是指在进行模型的查询和写入操作的时候触发的操作行为。

> 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 | 事件方法名 |
| --- | --- | --- |
| after_read | 查询后 | onAfterRead |
| before_insert | 新增前 | onBeforeInsert |
| after_insert | 新增后 | onAfterInsert |
| before_update | 更新前 | onBeforeUpdate |
| after_update | 更新后 | onAfterUpdate |
| before_write | 写入前 | onBeforeWrite |
| after_write | 写入后 | onAfterWrite |
| before_delete | 删除前 | onBeforeDelete |
| after_delete | 删除后 | onAfterDelete |
| before_restore | 恢复前 | onBeforeRestore |
| after_restore | 恢复后 | onAfterRestore |

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。

> 如果 `before_write` 、 `before_insert` 、 `before_update` 、 `before_delete` 事件方法中返回 `false` 或者抛出 `think\exception\ModelEventException` 异常的话，则不会继续执行后续的操作。

## 模型事件定义

最简单的方式是在模型类里面定义静态方法来定义模型的相关事件响应。

```
<?php
namespace app\model;

use think\Model;
use app\model\Profile;

class User extends Model
```

```
{
    public static function onBeforeUpdate($user)
    {
        if ('thinkphp' == $user->name) {
            return false;
        }
    }

    public static function onAfterDelete($user)
    {
        Profile::destroy($user->id);
    }
}
```

参数是当前的模型对象实例，支持使用依赖注入传入更多的参数。

# 模型关联

## 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::find(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\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class);
    }
}
```

> 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
<?php

namespace app\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：

```php
$user->userProfile;
$user->user_profile; // 建议使用
```

> 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， `6.0` 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 |
| `hasOneThrough` | 远程一对一 |
| `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。也可以使用完整命名空间定义，例如：

```
<?php

namespace app\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class);
    }
}
```

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\model;

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

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::find(1);
// 获取档案所属的用户名称
echo $profile->user->name;
```

如果是数据集查询的话，关联获取的用法如下：

```
// 获取档案实例
$profiles = Profile::where('id', '>', 1)->select();
foreach($profiles as $profile) {
    // 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\model;

use think\Model;

class User extends Model
{
    public function book()
    {
        return $this->hasMany(Book::class)->order('pub_time');
    }
}
```

> 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

### 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\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class);
    }
}
```

`hasOne` 方法的参数包括：

> hasOne('关联模型类名', '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）：关联模型类名
- 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 |
| --- | --- |
| 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` |

> 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对象名（不含命名空间）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::find(1);
// 输出Profile关联模型的email属性
echo $user->profile->email;
```

默认情况下， 我们使用的是 `user_id` 作为外键关联，如果不是的话则需要在关联定义的时候指定，例如：

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class, '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` 。

## 根据关联数据查询

可以根据关联条件来查询当前模型对象数据，例如：

```
// 查询用户昵称是think的用户
// 注意第一个参数是关联方法名（不是关联模型名）
$users = User::hasWhere('profile', ['nickname'=>'think'])->select();

// 可以使用闭包查询
$users = User::hasWhere('profile', function($query) {
    $query->where('nickname', 'like', 'think%');
})->select();
```

## 预载入查询

可以使用预载入查询解决典型的 `N+1` 查询问题，使用：

```
$users = User::with('profile')->select();
foreach ($users as $user) {
    echo $user->profile->name;
}
```

上面的代码使用的是 `IN` 查询，只会产生2条SQL查询。

如果要对关联模型进行约束，可以使用闭包的方式。

```
$users = User::with(['profile'  => function($query) {
    $query->field('id,user_id,name,email');
}])->select();
foreach ($users as $user) {
    echo $user->profile->name;
}
```

`with` 方法可以传入数组，表示同时对多个关联模型（支持不同的关联类型）进行预载入查询。

```
$users = User::with(['profile','book'])->select();
foreach ($users as $user) {
    echo $user->profile->name;
    foreach($user->book as $book) {
        echo $book->name;
    }
}
```

如果需要使用 `JOIN` 方式的查询，直接使用 `withJoin` 方法，如下：

```
$users = User::withJoin('profile')->select();
foreach ($users as $user) {
    echo $user->profile->name;
}
```

`withJoin` 方法默认使用的是 `INNER JOIN` 方式，如果需要使用其它的，可以改成

```
$users = User::withJoin('profile', 'LEFT')->select();
foreach ($users as $user) {
    echo $user->profile->name;
}
```

需要注意的是 `withJoin` 方式不支持嵌套关联，通常你可以直接传入多个需要关联的模型。

如果需要约束关联字段，可以使用下面的简便方法。

```
$users = User::withJoin([
    'profile'   =>  ['user_id', 'name', 'email']
])->select();
foreach ($users as $user) {
    echo $user->profile->name;
```

```
}
```

## 关联保存

```
$user = User::find(1);
// 如果还没有关联数据 则进行新增
$user->profile()->save(['email' => 'thinkphp']);
```

系统会自动把当前模型的主键传入 `Profile` 模型。

和新增一样使用 `save` 方法进行更新关联数据。

```
$user = User::find(1);
$user->profile->email = 'thinkphp';
$user->profile->save();
// 或者
$user->profile->save(['email' => 'thinkphp']);
```

## 定义相对关联

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\model;

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

`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）：关联模型类名
- 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\model;

use think\Model;

class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class, 'uid');
    }
}
```

我们就可以根据档案资料来获取用户模型的信息

```
$profile = Profile::find(1);
// 输出User关联模型的属性
echo $profile->user->account;
```

## 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\model;

use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class, 'uid')->bind(['nickname', 'email'])
;
    }
}
```

或者指定绑定属性别名

```
<?php
namespace app\model;
```

```
use think\Model;

class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class, 'uid')->bind([
                'email',
                'truename'  => 'nickname',
            ]);
    }
}
```

然后使用关联预载入查询的时候，可以使用

```
$user = User::with('profile')->find(1);
// 直接输出Profile关联模型的绑定属性
echo $user->email;
echo $user->truename;
```

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。

也可以使用动态绑定关联属性，可以使用

```
$user = User::find(1)->bindAttr('profile',['email','nickname']);
// 输出Profile关联模型的email属性
echo $user->email;
echo $user->nickname;
```

同样支持指定属性别名

```
$user = User::find(1)->bindAttr('profile',[
    'email',
    'truename'  => 'nickname',
]);
// 输出Profile关联模型的email属性
echo $user->email;
echo $user->truename;
```

## 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::find(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::with('content')->find(1);
// 删除当前及关联模型
$blog->together(['content'])->delete();
```

# 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）：关联模型类名
- 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\model;

use think\Model;

class Article extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

同样，也可以定义外键的名称

```
<?php
namespace app\model;

use think\Model;

class Article extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class,'art_id');
    }
}
```

## 关联查询

我们可以通过下面的方式获取关联数据

```
$article = Article::find(1);
// 获取文章的所有评论
dump($article->comments);
// 也可以进行条件搜索
dump($article->comments()->where('status',1)->select());
```

## 根据关联条件查询

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();
```

如果需要更复杂的关联条件查询，可以使用

```
$where = Comment::where('status',1)->where('content', 'like', '%think%');
$list = Article::hasWhere('comments', $where)->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\model;

use think\Model;
```

```
class Comment extends Model
{
    public function article()
    {
        return $this->belongsTo(Article::class);
    }
}
```

## 关联删除

在删除文章的同时删除下面的评论

```
$article = Article::with('comments')->find(1);
$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\model;

use think\Model;

class City extends Model
{
    public function topics()
    {
        return $this->hasManyThrough(Topic::class, User::class);
    }
}
```

> 远程一对多关联，需要同时存在 `Topic` 和 `User` 模型，当前模型和中间模型的关联关系可以是一对一或者一对多。

`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','当前模型主键','中间模型主键');

- 关联模型（必须）：关联模型类名
- 中间模型（必须）：中间模型类名
- 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`
- 当前模型主键：一般会自动获取也可以指定传入
- 中间模型主键：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

```
$city = City::find(1);
// 获取同城的所有话题
dump($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

## 根据关联条件查询

如果需要根据关联条件来查询当前模型，可以使用

```
$list = City::hasWhere('topics', ['status' => 1])->select();
```

更复杂的查询条件可以使用

```
$where = Topic::where('status', 1)->where('title', 'like', '%think%');
$list = City::hasWhere('topics',$where)->select();
```

# 远程一对一

远程一对一关联用于定义有跨表的一对一关系，例如：

- 每个用户有一个档案
- 每个档案有一个档案卡
- 用户和档案卡之间并无关联

## 关联定义

就可以直接通过远程一对一关联获取每个用户的档案卡， `User` 模型定义如下：

```
<?php
namespace app\model;

use think\Model;

class User extends Model
{
    public function card()
    {
        return $this->hasOneThrough(Card::class,Profile::class);
    }
}
```

远程一对一关联，需要同时存在 `Card` 和 `Profile` 模型。

`hasOneThrough` 方法的参数如下：

> hasOneThrough('关联模型', '中间模型', '外键', '中间表关联键','当前模型主键','中间模型主键');

- 关联模型（必须）：关联模型类名
- 中间模型（必须）：中间模型类名
- 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`
- 当前模型主键：一般会自动获取也可以指定传入
- 中间模型主键：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

```php
$user = User::find(1);
// 获取用户的档案卡
dump($user->card);
```

# 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\model;

use think\Model;

class User extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Role::class, 'access');
    }
}
```

`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）：关联模型类名
- 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::class, Access::class);
    }
```

中间表模型类必须继承 `think\model\Pivot` ，例如：

```
<?php
namespace app\model;

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::find(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::find(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::find(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::find(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::find(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\model;

use think\Model;

class Role extends Model
{
    public function users()
    {
        return $this->belongsToMany(User::class, Access::class);
    }
}
```

# 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
<?php
namespace app\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::class, '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）：关联的模型类名

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（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\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::class, '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\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::find(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::find(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\model;

use think\Model;

class Member extends Model
{
    /**
     * 获取用户的头像
     */
    public function avatar()
    {
        return $this->morphOne(Avatar::class, 'imageable');
    }
}
```

团队模型：

```
<?php
namespace app\model;

use think\Model;

class Team extends Model
{
    /**
     * 获取团队的头像
     */
    public function avatar()
    {
        return $this->morphOne(Avatar::class, '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）：关联的模型类名。
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（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\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：

```
$list = User::select([1,2,3]);
foreach($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 `withJoin` 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]);
```

> `with` 方法只能调用一次，请不要多次调用，如果需要对多个关联模型预载入使用数组即可。

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);
}
```

如果要指定属性查询，可以使用：

```
$list = User::field('id,name')->with(['profile' => function($query){
    $query->field('user_id,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);
}
```

> 记得指定属性的时候一定要包含关联键。

对于一对多关联来说，如果需要设置返回的关联数据数量，可以使用 `withLimit` 方法。

```
Article::with(['comments' => function(Relation $query) {
    $query->order('create_time', 'desc')->withLimit(3);
}])->select();
```

> 注意这里的类型约束必须使用 `think\model\Relation` ，因为 `withLimit` 方法是关联类才有的方法

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]);
```

和下面的方法是等效的：

```
$list = User::with(['user_profile','user_book'])->select([1,2,3]);
```

区别在于你获取关联数据的时候必须和传入的关联名称保持一致。

```
$user = User::with(['userProfile'])->find(1);
dump($user->userProfile);
```

```
$user = User::with(['user_profile'])->find(1);
dump($user->user_profile);
```

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，可以使用 `withJoin` 方法。

```
$list = User::withJoin(['profile' => function(Relation $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::select([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);
}
```

## 关联预载入缓存

关联预载入可以支持查询缓存，例如：

```
$list = User::with(['profile'])->withCache(30)->select([1,2,3]);
```

表示对关联数据缓存30秒。

如果你有多个关联数据，也可以仅仅缓存部分关联

```
$list = User::with(['profile', 'book'])->withCache(['profile'],30)->select([1,2,
3]);
```

对于延迟预载入查询的话，可以在第二个参数传入缓存参数。

```php
// 查询数据集
$list = User::select([1,2,3]);
// 延迟预载入
$list->load(['cards'], 30);
```

# 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;
}
```

> 你必须给User模型定义一个名称是 `cards` 的关联方法。

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` ”为名称的动态属性来保存相关的关联统计数据。
可以通过数组的方式同时查询多个统计字段。

```
$list = User::withCount(['cards', 'phone'])->select([1,2,3]);
foreach($list as $user){
    // 获取用户关联关联统计
    echo $user->cards_count;
    echo $user->phone_count;
}
```

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;
}
```

> 关联统计暂不支持多态关联

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;
}
```

使用闭包的方式，如果需要自定义统计字段名称，可以使用

```
$list = User::withCount(['cards' => function($query, &$alias) {
    $query->where('status',1);
    $alias = '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在第二个字段传入统计字段名，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;
}
```

所有的关联统计方法可以多次调用，每次查询不同的关联统计数据。

```
$list = User::withSum('cards', 'total')
    ->withSum('score', 'score')
    ->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，新版仅内置了PHP原生模板引擎（主要用于内置的异常页面输出），如果需要使用其它的模板引擎需要单独安装相应的模板引擎扩展。
如果你需要使用 `think-template` 模板引擎，只需要安装 `think-view` 模板引擎驱动。

```
composer require topthink/think-view
```

> 视图相关的配置在配置目录的 `view.php` 配置文件中进行定义。

通常可以直接使用 `think\facade\View` 来操作视图。

# 模板变量

## 模板赋值

模板中的变量（除了一些系统变量外）必须先进行模板赋值后才能使用，可以使用 `assign` 方法进行全局模板变量赋值。

```
namespace app\controller;

use think\facade\View;

class Index
{
    public function index()
    {
        // 模板变量赋值
        View::assign('name','ThinkPHP');
        View::assign('email','thinkphp@qq.com');
        // 或者批量赋值
        View::assign([
            'name'  => 'ThinkPHP',
            'email' => 'thinkphp@qq.com'
        ]);
        // 模板输出
        return View::fetch('index');
    }
}
```

`assign` 方法赋值属于全局变量赋值，如果你需要单次赋值的话，可以直接在 `fetch` 方法中传入。

```
namespace app\controller;

use think\facade\View;

class Index
{
    public function index()
    {
        // 模板输出并变量赋值
        return View::fetch('index', [
            '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'
]);
```

助手函数的变量赋值也是当次模板渲染有效。

# 视图过滤

## 视图过滤

可以对视图的渲染输出进行过滤

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

use think\facade\View;

class Index
{
    public function index()
    {
        // 使用视图输出过滤
        return View::filter(function($content){
            return str_replace("\r\n",'<br/>',$content);
        })->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
<?php
namespace app\index\controller;

class Index
{
    public function index()
    {
        // 使用视图输出过滤
        return view()->filter(function($content){
            return str_replace("\r\n",'<br/>',$content);
        });
    }
}
```

# 模板渲染

## 模板路径

默认情况下，框架会自动定位你的模板文件路径，优先定位应用目录下的 `view` 目录，这种方式的视图目录下就是应用的控制器目录。

### 单应用模式

```
├─app
│   └─view（视图目录）
│     ├─index             index控制器目录
│     │  └─index.html     index模板文件
│     └─ ...              更多控制器目录
```

### 多应用模式

```
├─app
│  ├─app1 （应用1）
│  │   └─view（应用视图目录）
│  │     ├─index             index控制器目录
│  │     │  └─index.html     index模板文件
│  │     └─ ...              更多控制器目录
│  │
│  └─ app2...  （更多应用）
```

第二种方式是视图文件和应用类库文件完全分离，统一放置在根目录下的 `view` 目录。

### 单应用模式

```
├─view                  视图文件目录
│   ├─index             index控制器目录
│   │  └─index.html     index模板文件
│   └─ ...              更多控制器目录
```

### 多应用模式

如果是多应用模式的话，这种方式下 `view` 目录下面首先是应用子目录。

```
├─view                  视图文件目录
│  ├─index（应用视图目录）
│  │   ├─index             index控制器目录
│  │   │  └─index.html     index模板文件
│  │   └─ ...              更多控制器目录
```

如果你需要自定义 `view` 目录名称，可以通过设置 `view_dir_name` 配置参数。

```
'view_dir_name'    =>    'template',
```

## 模板渲染

模板渲染的最典型用法是直接使用 `fetch` 方法，不带任何参数：

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

use think\facade\View;

class Index
{
    public function index()
    {
        // 不带任何参数 自动定位当前操作的模板文件
        return View::fetch();
    }
}
```

表示系统会按照默认规则自动定位视图目录下的模板文件，其规则是：

```
控制器名（小写+下划线）/操作名.html
```

默认的模板文件名规则改为实际操作方法名的小写+下划线写法。但可以配置 `auto_rule` 参数的值来改变当前操作的自动渲染规则。

| auto_rule配置 | 自动定位规则 |
| --- | --- |
| 1 | 操作方法的小写+下划线 |
| 2 | 操作方法全部转换小写 |
| 3 | 保持和操作方法一致 |

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：

```
控制器（小写+下划线）_操作.html
```

如果没有按照模板定义规则来定义模板文件（或者需要调用其他控制器下面的某个模板），可以使用：

```
// 指定模板输出
return View::fetch('edit');
```

表示调用当前控制器下面的edit模板

```
return View::fetch('member/read');
```

表示调用Member控制器下面的read模板。

跨应用渲染模板

```
return View::fetch('admin@member/edit');
```

渲染输出不需要写模板文件的路径和后缀。这里面的控制器和操作并不一定需要有实际对应的控制器和操作，只是一个目录名称和文件名称而已，例如，你的项目里面可能根本没有Public控制器，更没有Public控制器的menu操作，但是一样可以使用

```
return View::fetch('public/menu');
```

输出这个模板文件。理解了这个，模板输出就清晰了。

支持从视图根目录开始读取模板，例如：

```
return View::fetch('/menu');
```

表示读取的模板是

```
menu.html
```

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 View::fetch('../template/public/menu.html');
```

这种方式需要带模板路径和后缀指定一个完整的模板文件位置，这里的 `../template/public` 目录是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 View::fetch('../template/public/menu.tpl');
```

只要 `../template/public/menu.tpl` 是一个实际存在的模板文件。

> 要注意模板文件位置是相对于应用的入口文件，而不是模板目录。

## 助手函数

可以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;

class Index
{
    public function index()
    {
        // 渲染模板输出
        return view('hello', ['name' => 'thinkphp']);
    }
}
```

## 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\facade\View;

class Index
{
    public function index()
    {
        // 直接渲染内容
        $content = '{$name}-{$email}';
        return View::display($content, ['name' => 'thinkphp', 'email' => 'think
php@qq.com']);
    }
}
```

渲染的内容中一样可以使用模板引擎的相关标签。

# 模板引擎

## 使用 `thinkTemplate` 模板引擎

新版框架默认只能支持PHP原生模板，如果需要使用 `thinkTemplate` 模板引擎，需要安装 `think-view` 扩展（该扩展会自动安装 `think-template` 依赖库）。

```
composer require topthink/think-view
```

### 配置文件

安装完成后，在配置目录的 `view.php` 文件中进行模板引擎相关参数的配置，例如：

```
return [
    // 模板引擎类型
    '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'     => '/',
    // 模板引擎普通标签开始标记
    'tpl_begin'     => '{',
    // 模板引擎普通标签结束标记
    'tpl_end'       => '}',
    // 标签库标签开始标记
    'taglib_begin' => '{',
    // 标签库标签结束标记
    'taglib_end'    => '}',
];
```

## 调用 `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：

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

use think\facade\View;

class Index
{
    public function index()
```

```
    {
        // 使用内置PHP模板引擎渲染模板输出
        return View::engine('php')->fetch();
    }
}
```

表示当前视图的模板文件使用原生php进行解析。

> 如果你需要动态改变模板引擎的参数，请使用视图类提供的 `config` 方法进行动态设置，而不要使用改变配置类参数的方式。

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

use think\facade\View;

class Index
{
    public function index()
    {
        // 改变当前操作的模板路径
        View::config(['view_path' => 'mypath']);
        return View::fetch();
    }
}
```

关于模板引擎的标签用法，可以参考这里。

# 视图驱动

## 视图驱动

默认的视图仅支持PHP原生模板，如果你需要扩展支持其它的模板引擎，可以实现一个ThinkPHP视图的模板引擎驱动，该驱动必须实现 `think\contract\TemplateHandlerInterface` 接口，包含下列方法。

```
interface TemplateHandlerInterface
{

    /**
     * 检测是否存在模板文件
     * @access public
     * @param  string $template 模板文件或者模板规则
     * @return bool
     */
    public function exists(string $template): bool;

    /**
     * 渲染模板文件
     * @access public
     * @param  string    $template 模板文件
     * @param  array     $data 模板变量
     * @return void
     */
    public function fetch(string $template, array $data = []): void;

    /**
     * 渲染模板内容
     * @access public
     * @param  string    $content 模板内容
     * @param  array     $data 模板变量
     * @return void
     */
    public function display(string $content, array $data = []): void;

    /**
     * 配置模板引擎
     * @access private
     * @param  array  $config 参数
     * @return void
     */
    public function config(array $config): void;

    /**
     * 获取模板引擎配置
     * @access public
```

```
     * @param  string  $name 参数名
     * @return void
     */
    public function getConfig(string $name);
}
```

# 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

新版的异常页面显示会自动判断当前的请求是否为Json请求，如果是JSON请求则采用JSON格式输出异常信息，否则按照HTML格式输出。

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'         => \think\facade\App::getAppPath() . 'template/excepti
on.tpl',
```

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。

## 异常处理接管

> 本着严谨的原则，框架会对任何错误（包括警告错误）抛出异常。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。

框架支持异常处理由开发者自定义类进行接管，需要在 `app` 目录下面的 `provider.php` 文件中绑定异常处理类，例如：

```
    // 绑定自定义异常处理handle类
    'think\exception\Handle'       => '\\app\\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 think\exception\Handle;
use think\exception\HttpException;
use think\exception\ValidateException;
use think\Response;
```

```
use Throwable;

class Http extends Handle
{
    public function render($request, Throwable $e): Response
    {
        // 参数验证错误
        if ($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($request, $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。

> 事实上，默认安装应用后，已经帮你内置了一个 `app\ExceptionHandle` 异常处理类，直接修改该类的相关方法即可完成应用的自定义异常处理机制。

> 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);
```

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。

## 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;


class Index
{
    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;


class Index
{
    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 `app.php` 配置文件中增加：

```
'http_exception_template'    =>  [
    // 定义404错误的模板文件地址
    404 =>  \think\facade\App::getAppPath() . '404.html',
    // 还可以定义其它的HTTP status
    401 =>  \think\facade\App::getAppPath() . '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。

> 日志遵循 PSR-3 规范,除非是实时写入的日志,其它日志都是在当前请求结束的时候统一写入的 所以不要在日志写入之后使用 exit 等中断操作会导致日志写入失败。



## 日志配置

日志的配置文件是配置文件目录下的 `log.php` 文件，系统在进行日志写入之前会读取该配置文件进行初始化。

新版的日志配置支持多通道，默认配置如下：

```
return [
    // 默认日志记录通道
    'default'      => 'file',
    // 日志记录级别
    'level'        => [],
    // 日志类型记录的通道 ['error'=>'email',...]
    'type_channel' => [],

    // 日志通道列表
```

```
    'channels'     => [
        'file' => [
            // 日志记录方式
            'type'         => 'File',
            // 日志保存目录
            'path'         => '',
            // 单文件日志写入
            'single'       => false,
            // 独立日志级别
            'apart_level' => [],
            // 最大日志文件数量
            'max_files'    => 0,
        ],
        // 其它日志通道配置
    ],
];
```

可以添加多个日志通道，每个通道可以设置不同的日志类型。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` ，日志类型使用 `type` 参数配置即可。

> 如果是自定义驱动， `type` 的值则为自定义驱动的类名（包含命名空间）

日志的全局配置参数包含：

| 参数 | 描述 |
| --- | --- |
| default | 默认的日志通道 |
| level | 允许记录的日志级别 |
| type_channel | 日志类型记录的通道 |

> 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 |
| single | 是否单一文件日志 |
| max_files | 最大日志文件数（超过自动清理 ） |
| format | 日志输出格式 |

| realtime_write | 是否实时写入 |
| --- | --- |

> 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。

## 日志写入

### 手动记录

一般情况下，系统的错误日志记录是自动的，如果需要记录应用的业务日志或者额外的日志信息，就需要手动记录日志信息，Log类主要提供了2个方法用于记录日志。

| 方法 | 描述 |
| --- | --- |
| record() | 记录日志信息到内存 |
| write() | 实时写入一条日志信息 |

系统在请求结束后会自动调用 `Log::save` 方法统一进行日志信息写入。

`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');
```

你可以在日志通道配置开启实时写入，每次记录日志信息的时候就会实时写入。

```
'file' => [
    // 日志记录方式
    'type'         => 'File',
    // 日志保存目录
    'path'         => '',
    // 单文件日志写入
    'single'       => false,
    // 独立日志级别
    'apart_level' => [],
    // 最大日志文件数量
    'max_files'    => 0,
```

```
    // 日志处理
    'processor'    => null,
    // 实时写入
    'realtime_write'    =>    true,
],
```

> 为避免内存溢出，在命令行下面执行的话日志信息会自动实时写入。

## 关闭日志

要关闭日志功能，可以调用 `Log::close()` 方法关闭本次请求的日志写入。

```
// 关闭当前日志写入
Log::close();
```

## 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');
```

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');
```

事实上，你可以增加自定义的日志类型，例如：

```
Log::diy('这是一个自定义日志类型');
```

也支持指定级别日志的写入，需要配置信息：

```
return [
    // 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 [
    'default'      => 'file',
    'channels'     =>    [
        'file'     =>    [
            '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。

> 如果 `apart_level` 设置为 `true` ，则表示所有的日志类型都会独立记录。

## 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 [
    'default'      => 'file',
    'channels'     =>    [
        'file'     =>    [
```

```
            'type'          => 'file',
            '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 [
    'default'      => 'file',
    'channels'    =>    [
        'file'    =>    [
            'type'          => 'file',
            '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。

> 单文件日志也支持 max_files 参数设置，因为单文件日志同样会生成多个日志备份文件而导致日志文件数据过大。

## 写入处理

日志支持写入回调处理，通过事件的方式处理。

```
Event::listen('think\event\LogWrite', function($event) {
    if('file' == $event->channel) {
        $event->log['info'][] = 'test info';
    }
});
```

## 格式化日志信息

系统提供了两个参数用于日志信息的格式化，第一个是用于自定义时间显示格式的 `time_format` ，第二个是调整日志输出格式的 `format` 参数。

```
return  [
```

```
    'default'      => 'file',
    'channels'    =>    [
        'file'    =>    [
            'type'          => 'file',
            'json'          =>  true
            'file_size'     =>  1024*1024*10,
            'time_format'   =>    'Y-m-d H:i:s',
            'format'        =>    '[%s][%s]:%s',
        ],
    ],
];
```

## 清空日志

一旦执行 `save` 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();
```

在清空日志方法之前，你可以使用 `getLog` 方法获取内存中的日志。

```
// 获取全部日志
$logs = Log::getLog();
```

> 日志清空仅仅是清空内存中的日志。

## 日志自动清理

文件类型的日志支持自动清理。可以设置 `max_files` 参数，超过数量的最早日志将会自动删除。

例如，下面设置日志最多保存数量为30个

```
return [
    'default'      => 'file',
    'channels'    =>    [
        'file'    =>    [
            'type'          => 'file',
            'max_files' =>  30,
            'file_size'     =>  1024*1024*10,
        ],
    ],
];
```

> 设置 `max_files` 参数后，日志文件将不会分日期子目录存放。

## JSON格式日志

可以支持 `JSON` 格式记录文件日志，更加方便一些第三方日志分析工具进行日志分析。

在日志配置文件中，添加

```
return [
    'default'      => 'file',
    'channels'    =>    [
        'file'    =>    [
            'type'          => 'file',
            'json'          =>  true
            'file_size'     =>  1024*1024*10,
        ],
    ],
];
```

即可开启 `JSON` 格式记录，CLI命令行的日志记录同样有效。

使用JSON格式记录后，每次请求是一行JSON数据，但如果使用 `Log::write` 记录的日志是例外的单独一行JSON数据。

## 日志通道

你可以配置不同的日志通道，并且把不同的日志记录到不同的通道。

```
Log::channel('email')->info('一条测试日志');
Log::channel('socket')->error('记录错误日志');
```

你可以配置不同的日志类型，记录到不同的日志通道，这样在记录日志的时候会自动选择对应的通道写入。

```
return [
    'default'    =.    ‘file',
    'type_channel'    =>    [
        'error'    =>    'email',
        'sql'    =>    'sql',
    ],
    'channels'    =>    [
        'file'    =>    [
            'type'    =>    'file',
        ],
        'email'    =>    [
            'type'    =>    'email',
        ],
        'sql'    =>    [
            'type'    =>    'sql',
        ],
    ],
];
```

表示如果是 `error` 日志和 `sql` 日志，会分别记录到指定的通道。同时你还需要在日志配置文件中，添加 `email` 和 `sql` 日志通道的配置。核心只有 `file` 日志类型，其它的可能需要自己扩展或者安装扩展。
如果需要获取内存中的通道日志信息，可以使用

```
// 获取某个日志通道的日志
$error = Log::getLog('file');
```

可以单独关闭某个通道的日志写入，只需要把日志通道的 `close` 配置参数设置为 `true` ，或者使用方法关闭。

```
Log::close('file');
```

可以单独清空某个通道的日志（如果没有开启实时写入的话）

```
Log::clear('file');
```

## 自定义驱动

如果需要自定义日志驱动，你需要实现 `think\contract\LogHandlerInterface` 接口。

```
interface LogHandlerInterface
{
    /**
     * 日志写入接口
     * @access public
     * @param  array $log 日志信息
     * @return bool
     */
    public function save(array $log): bool;

}
```

# 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，可以修改环境变量 `APP_DEBUG` 开启调试模式，上线部署后切换到部署模式。
本地开发的时候可以在应用根目录下面定义 `.env` 文件。

> 通过 `create-project` 默认安装的话， 会在根目录自带一个 `.example..env` 文件，你可以直接更名为 `.env` 文件。

`.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
// ...
```

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；
- 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。最新版本页面Trace功能已经不再内置在核心，但默认安装的时候会自动安装 `topthink/think-trace` 扩展，所以你可以在项目里面直接使用。

如果部署到服务器的话，你可以通过下面方式安装

```
composer install --no-dev
```

就不会安装页面Trace扩展。

## 使用

> 页面Trace功能仅在调试模式下有效

安装页面Trace扩展后，如果开启调试模式并且运行后有页面有输出的话，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',
    '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',
    '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：


如果需要更改页面Trace输出的样式，可以自定义模板文件（可以复制内置模板文件 `vendor/topthink/think-trace/src/tpl/page_trace.tpl` 的内容），然后配置 `file` 参数指定模板文件位置。

```
'file'    =>    'app\trace\page_trace.html',
```

## 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 `tabs` 设置。

> 由于页面Trace功能是通过中间件来执行，所以命令行下面不会显示任何的页面Trace信息

# 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：

基本 文件 流程 错误 **SQL** 调试

```
[ DB ] CONNECT:[ UseTime:0.001285s ] mysql:dbname=demo;host=127.0.0.1;charset=utf8
[ SQL ] SHOW COLUMNS FROM `user` [ RunTime:0.001620s ]
[ SQL ] SELECT * FROM `user` WHERE `id` = 4 LIMIT 1 [ RunTime:0.000695s ]
[ SQL ] SHOW COLUMNS FROM `profile` [ RunTime:0.001021s ]
[ SQL ] INSERT INTO `profile` (`truename` , `birthday` , `address` , `email` , `user_id`) VALUES ('top' , 123456 , 'ddd' , 'ddd' , 4) [ RunTime:0.000535s ]
[ SQL ] SELECT * FROM `profile` WHERE `user_id` = 4 LIMIT 1 [ RunTime:0.000516s ]
```

## 查看SQL日志

如果开启了数据库的日志监听（ `trigger_sql` )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。

> 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 ]
```

如果需要增加额外的SQL监听，可以使用

```
Db::listen(function($sql, $runtime, $master) {
    // 进行监听处理
});
```

监听方法支持三个参数，依次是执行的SQL语句，运行时间（秒），以及主从标记（如果没有开启分布式的话，该参数为null，否则为布尔值）。

## 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::find(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

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。

用法和PHP内置的 `var_dump` 一致：

```
dump($var1, ...$varN)
```

使用示例：

```
$blog = Db::name('blog')->where('id', 3)->find();
$user = User::find();
dump($blog, $user);
```

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();
$user = User::find();
halt($blog, $user);
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。

> 为了方便查看，某些核心对象由于定义了 `__debugInfo` 方法，因此在 `dump` 输出的时候属性可能做了简化。

# 远程调试

`ThinkPHP` 提供了 `Socket` 日志驱动用于本地和远程调试。
首先需要安装 `think-socketlog` 扩展

```
composer require topthink/think-socketlog
```

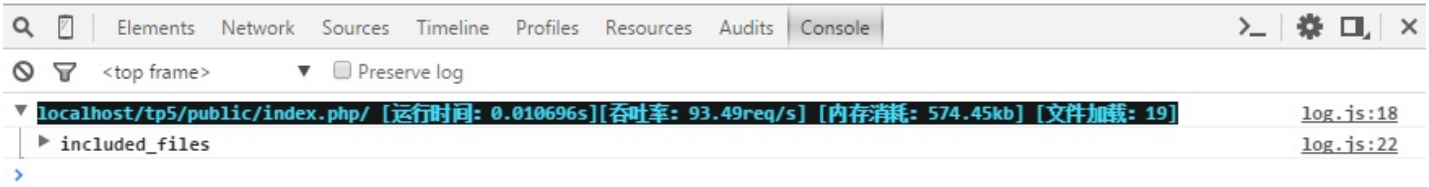
## `Socket` 调试

只需要在 `log.php` 配置文件中设置如下：

```
return [
    'type'                  => 'SocketLog',
    '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推荐使用验证器，可以在控制器中使用 `validate` 助手函数（或者封装验证方法）进行验证。

## 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\validate\User` 验证器类用于 `User` 的验证。

```
namespace app\validate;

use think\Validate;

class User extends Validate
{
    protected $rule = [
        'name'  =>  'require|max:25',
        'email' =>  'email',
    ];

}
```

> 可以使用下面的指令快速生成 `User` 验证器。
>
> ```
> php think make:validate User
> ```

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\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：

```
<?php
namespace app\controller;

use app\validate\User;
use think\exception\ValidateException;

class Index
{
    public function index()
    {
        try {
            validate(User::class)->check([
                'name'  => 'thinkphp',
                'email' => 'thinkphp@qq.com',
            ]);
        } catch (ValidateException $e) {
            // 验证失败 输出错误信息
            dump($e->getError());
        }
    }
}
```

## 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：

```
<?php
namespace app\controller;

use app\validate\User;
use think\exception\ValidateException;
```

```
class Index
{
    public function index()
    {
    try {
        $result = validate(User::class)->batch(true)->check([
            'name'  => 'thinkphp',
            'email' => 'thinkphp@qq.com',
            ]);

        if (true !== $result) {
            // 验证失败 输出错误信息
            dump($result);
        }
    } catch (ValidateException $e) {
            // 验证失败 输出错误信息
            dump($e->getError());
        }
    }
}
```

## 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：

```
<?php
namespace app\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\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\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。独立验证通常使用 `Facade` 或者自己实例化验证类。

```
$validate = \think\facade\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。
我们把上面的验证代码改为

```
use think\facade\Validate;
use think\validate\ValidateRule as Rule;

$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：

```
$validate = Validate::rule([
    'name'  => function($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。

```
$validate = Validate::rule([
    'name'  => function($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

## 全局扩展

你可以在扩展包或者应用里面全局注册验证规则，使用方法

```
Validate::maker(function($validate) {
    $validate->extend('extra', 'extra_validate_callback');
});
```

# 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\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\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\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，可以定义 `message` 属性：

```
namespace app\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\validate\User;
$result = $validate->check($data);

if(!$result){
    echo $validate->getError();
}
```

会输出 `年龄必须在1~120之间` 。

错误信息可以支持数组定义，并且通过JSON方式传给前端。

```
namespace app\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' => ['code' => 1001, 'msg' => '名称必须'],
      'name.max'     => ['code' => 1002, 'msg' => '名称最多不能超过25个字符'],
      'age.number'   => ['code' => 1003, 'msg' => '年龄必须是数字'],
      'age.between'  => ['code' => 1004, 'msg' => '年龄必须在1~120之间'],
      'email'        => ['code' => 1005, 'msg' =>'邮箱格式错误'],
    ];
}
```

## 使用多语言

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\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

> 验证场景仅针对验证器有效，独立验证不存在验证场景的概念

验证器支持定义场景，并且验证不同场景的数据，例如：

```
namespace app\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',
];

try {
    validate(app\validate\User::class)
        ->scene('edit')
        ->check($data);
} catch (ValidateException $e) {
```

```
    // 验证失败 输出错误信息
    dump($e->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：

- 注意：场景名不区分大小写，且在调用的时候不能将驼峰写法转为下划线

```
namespace app\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\validate\User` 进行自动验证，并且使用 `edit` 验证场景：

```
Route::post('hello/:id', 'index/hello')
    ->validate(\app\validate\User::class,'edit');
```

或者不使用验证器而直接传入验证规则

```
Route::post('hello/:id', 'index/hello')
    ->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')
    ->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
- 其它验证

> 验证规则严格区分大小写

## 格式验证类

格式验证类的验证规则如果在使用静态方法调用的时候需要加上 `is` （以 `number` 验证为例，需要使用 `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纯数字（采用 `ctype_digit` 验证，不包含负数和小数点），例如：

```
'num'=>'number'
```

### integer

验证某个字段的值是否为整数（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纯字母，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'
```

### cntrl

验证某个字段的值只能是控制字符（换行、缩进、空格），例如：

```
'name'=>'cntrl'
```

### graph

验证某个字段的值只能是可打印字符（空格除外），例如：

```
'name'=>'graph'
```

### print

验证某个字段的值只能是可打印字符（包括空格），例如：

```
'name'=>'print'
```

### lower

验证某个字段的值只能是小写字符，例如：

```
'name'=>'lower'
```

### upper

验证某个字段的值只能是大写字符，例如：

```
'name'=>'upper'
```

### space

验证某个字段的值只能是空白字符（包括缩进，垂直制表符，换行符，回车和换页字符），例如：

```
'name'=>'space'
```

### xdigit

验证某个字段的值只能是十六进制字符串，例如：

```
'name'=>'xdigit'
```

### 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

## 其它验证

### token:表单令牌名称

表单令牌验证

### 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'
```

### requireWithout:field

验证某个字段没有值的时候必须，例如：

```
// mobile和phone必须输入一个
'mobile' => 'requireWithout:phone',
'phone'  => 'requireWithout:mobile'
```

### 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。

# 表单令牌

## 添加令牌 `Token` 验证

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="{:token()}" />
```

也可以直接使用

```
{:token_field()}
```

默认的令牌Token名称是 `__token__` ，如果需要自定义名称及令牌生成规则可以使用

```
{:token_field('__hash__', 'md5')}
```

第二个参数表示token的生成规则，也可以使用闭包。

如果你没有使用默认的模板引擎，则需要自己生成表单隐藏域

```
namespace app\controller;

use think\Request;
use think\facade\View;

class Index
{
    public function index(Request $request)
    {
        $token = $request->buildToken('__token__', 'sha1');
        View::assign('token', $token);
        return View::fetch();
    }
}
```

然后在模板表单中使用：

```
<input type="hidden" name="__token__" value="{$token}" />
```

## AJAX提交

如果是AJAX提交的表单，可以将 `token` 设置在 `meta` 中

```
<meta name="csrf-token" content="{:token()}">
```

或者直接使用

```
{:token_meta()}
```

然后在全局Ajax中使用这种方式设置 `X-CSRF-Token` 请求头并提交：

```
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});
```

## 路由验证

然后在路由规则定义中，使用

```
Route::post('blog/save','blog/save')->token();
```

如果自定义了 `token` 名称，需要改成

```
Route::post('blog/save','blog/save')->token('__hash__');
```

令牌检测如果不通过，会抛出 `think\exception\ValidateException` 异常。

## 控制器验证

如果没有使用路由定义，可以在控制器里面手动进行令牌验证

```
namespace app\controller;

use think\exception\ValidateException;
use think\Request;

class Index
{
    public function index(Request $request)
    {
        $check = $request->checkToken('__token__');

        if(false === $check) {
```

```
            throw new ValidateException('invalid token');
        }

        // ...
    }
}
```

提交数据默认获取 `post` 数据，支持指定数据进行 `Token` 验证。

```
namespace app\controller;

use think\exception\ValidateException;
use think\Request;

class Index
{
    public function index(Request $request)
    {
        $check = $request->checkToken('__token__', $request->param());

        if(false === $check) {
            throw new ValidateException('invalid token');
        }

        // ...
    }
}
```

## 使用验证器验证

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__` （假设是 `__hash__` ），验证器中需要改为：

```
protected $rule = [
        'name'  =>  'require|max:25|token:__hash__',
        'email' =>  'email',
    ];
```

# 注解验证

## 注解验证器

ThinkPHP支持使用注解方式定义路由和验证，需要安装额外的扩展：

```
composer require topthink/think-annotation
```

然后可以直接在控制器类的方法注释中定义，例如：

```
<?php
namespace app\controller;

use think\annotation\Route;
use think\annotation\route\Validate;
use app\validate\IndexValidate;

class Index
{
    /**
     * @Validate(IndexValidate::class,scene="create",batch="true")
     * @return mixed
     * @Route("hello")
     */
    public function hello()
    {
        return 'hello, TP6 Annotation  Validate';
    }
}
```

`@Route("hello/:name")` 和 `@Validate(IndexValidate::class)` 就是注解路由和验证器的内容，请务必注意注释的规范，不能在注解路由里面使用单引号，否则可能导致注解路由解析失败，可以利用IDE生成规范的注释。如果你使用 `PHPStorm` 的话，建议安装 `PHP Annotations` 插件：https://plugins.jetbrains.com/plugin/7320-php-annotations，可以支持注解的自动完成。

然后需要声明上面引用验证器类，例如：

```
<?php

namespace app\validate;

use think\Validate;

class IndexValidate extends Validate
```

```
{
    protected $rule = [
        'name' => 'require'
    ];

   protected $message = [
       'name.require' => '姓名必须填写',
   ];

    protected $scene = [
        'create' => ['name'],
    ];
}
```

> 该方式定义的路由在调试模式下面实时生效，部署模式则在第一次访问的时候生成注解缓存。

### 注解验证器参数说明

> `value` 参数，可以不写 `value=` ，直接抒写值就可以了

| 参数名 | 参数类型 | 参数默认值 | 参数说明 |
| --- | --- | --- | --- |
| value | string | | 验证器 |
| scene | string | | 验证场景 |
| batch | bool | true | 统一验证：true=是，flase=不是 |
| message | array | [] | 错误内容 |

错误访问示例：

```
http://127.0.0.1:8000/hello
```

页面输出


正确访问示例：

```
http://127.0.0.1:8000/hello?name=zhans
```

页面输出


hello, TP6 Annotation Validate

# 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。

> ThinkPHP的缓存类遵循 `PSR-16` 规范。

## 设置

全局的缓存配置直接修改配置目录下面的 `cache.php` 文件。
新版的缓存支持多通道，你可以事先定义好所有的缓存类型及配置参数，然后在使用的时候可以随时切换。默认使用的是文件缓存类型，你可以添加 `redis` 缓存支持，例如：

```
return [
    'default'    =>    'file',
    'stores'    =>    [
        // 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',
        ],
    ],
];
```

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） |
| serialize | 缓存序列化和反序列化方法 |

> 如果是自定义驱动， `type` 的值则为自定义驱动的类名（包含命名空间）

## 使用

### 设置缓存

设置缓存有效期

```
// 缓存在3600秒之后过期
Cache::set('name', $value, 3600);
```

可以使用 `DateTime` 对象设置过期时间

```
Cache::set('name', $value, new DateTime('2019-10-01 12:00:00'));
```

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：

```
Cache::set('name', 1);
// name自增（步进值为1）
Cache::inc('name');
// name自增（步进值为3）
Cache::inc('name',3);
```

> 只能对数字或者浮点型数据进行自增和自减操作。

### 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：

```
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：

```
Cache::get('name','');
```

表示如果 `name` 值不存在，则返回空字符串。

### 追加一个缓存数据

如果缓存数据是一个数组，可以通过 `push` 方法追加一个数据。

```
Cache::set('name', [1,2,3]);
Cache::push('name', 4);
Cache::get('name'); // [1,2,3,4]
```

### 删除缓存

```
Cache::delete('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('start_time', time());
```

如果start_time缓存数据不存在，则会设置缓存数据为当前时间。

第二个参数可以使用闭包方法获取缓存数据，并支持依赖注入。

```
Cache::remember('start_time', function(Request $request){
    return $request->time();
});
```

remember方法的第三个参数可以设置缓存的有效期。

## 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');

// 清除tag标签的缓存数据
Cache::tag('tag')->clear();
```

并支持同时指定多个缓存标签操作

```
Cache::tag(['tag1', 'tag2'])->set('name1', 'value1');
Cache::tag(['tag1', 'tag2'])->set('name2', 'value2');

// 清除多个标签的缓存数据
Cache::tag(['tag1','tag2'])->clear();
```

可以追加某个缓存到标签

```
Cache::tag('tag')->append('name3');
```

## 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：

```
// 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：

```
// 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);
// 返回缓存对象实例
$cache = cache();
```

## 跨应用缓存

默认情况下，文件缓存数据是区分不同应用的，如果你希望缓存跨应用，可以设置一个统一的数据缓存 `path` 目录。

## 切换缓存类型

没有指定缓存类型的话，默认读取的是 `default` 缓存配置，可以动态切换

```
// 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::store('redis')->get('name');

// 切换到文件缓存
Cache::store('default')->set('name','value',3600);
Cache::store('default')->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();
```

## 自定义驱动

如果需要自定义缓存驱动，需要继承 `think\cache\Driver` 类，并且实现 `think\contract\CacheHandlerInterface` 接口。

```
interface CacheHandlerInterface
{
    /**
     * 判断缓存
     * @access public
     * @param  string $name 缓存变量名
     * @return bool
     */
    public function has($name): bool;

    /**
     * 读取缓存
     * @access public
     * @param  string $name 缓存变量名
     * @param  mixed  $default 默认值
     * @return mixed
```

```
     */
    public function get($name, $default = false);

    /**
     * 写入缓存
     * @access public
     * @param  string            $name 缓存变量名
     * @param  mixed             $value  存储数据
     * @param  integer|\DateTime $expire  有效时间（秒）
     * @return bool
     */
    public function set($name, $value, $expire = null): bool;

    /**
     * 自增缓存（针对数值缓存）
     * @access public
     * @param  string    $name 缓存变量名
     * @param  int       $step 步长
     * @return false|int
     */
    public function inc(string $name, int $step = 1);

    /**
     * 自减缓存（针对数值缓存）
     * @access public
     * @param  string    $name 缓存变量名
     * @param  int       $step 步长
     * @return false|int
     */
    public function dec(string $name, int $step = 1);

    /**
     * 删除缓存
     * @access public
     * @param  string $name 缓存变量名
     * @return bool
     */
    public function delete($name): bool;

    /**
     * 清除缓存
     * @access public
     * @return bool
     */
    public function clear(): bool;

}
```

使用自定义驱动后，只需要配置缓存 `type` 的值为该驱动类名（包含命名空间）即可。

# Session

## 概述

可以直接使用 `think\facade\Session` 类操作 `Session` 。

> 新版本不支持操作原生 `$_SESSION` 数组和所有 `session_` 开头的函数，只能通过 `Session` 类（或者助手函数）来操作。会话数据统一在当前请求结束的时候统一写入 所以不要在 `session` 写入操作之后执行 `exit` 等中断操作,否则会导致 `Session` 数据写入失败。

`6.0` 的 `Session` 类可以很好的支持诸如 `Swoole` / `Workerman` 等环境。

## 开启Session

`Session` 功能默认是没有开启的（API应用通常不需要使用 `Session` ），如果你需要使用 `Seesion` ，需要在全局的中间件定义文件中加上下面的中间件定义：

```
'think\middleware\SessionInit'
```

> 如果是多应用模式，并且你只是用于部分应用，那么也可以在应用中间件定义文件中单独开启。

## Session初始化

系统会自动按照 `session.php` 配置的参数自动初始化 `Session` 。
默认支持的 `session` 设置参数包括：

| 参数 | 描述 |
| --- | --- |
| type | session类型（ `File` 或者 `Cache` ） |
| store | 当type设置为cache类型的时候指定存储标识 |
| expire | session过期时间（秒）必须大于0 |
| var_session_id | 请求session_id变量名 |
| name | session_name |
| prefix | session前缀 |
| serialize | 序列化方法 |

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');
```

会话数据保存（请求结束）的时候会自动序列化，并在读取的时候自动反序列化，默认使用 `serialize` / `unserialize` 进行序列化操作，你可以自定义序列化机制。

例如在配置文件中设置为使用 `JSON` 序列化：

```
'serialize'    =>    ['json_encode', 'json_decode'],
```

> 尽量避免把对象保存到 `Session` 会话

## 基础用法

### 赋值

```
Session::set('name', 'thinkphp');
```

### 判断是否存在

```
Session::has('name');
```

### 取值

```
// 如果值不存在，返回null
Session::get('name');
// 如果值不存在，返回空字符串
Session::get('name', '');
// 获取全部数据
Session::all();
```

### 删除

```
Session::delete('name');
```

### 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

```
Session::clear();
```

### 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();
```

> 注意， `Session` 写入数据的操作会在请求结束的时候统一进行本地化存储，所以不要在写入 `Session` 数据之后使用exit等中断操作，可能会导致 `Session` 没有正常写入。

## 多级数组

支持 `session` 的多级数组操作，例如：

```
// 赋值
Session::set('name.item','thinkphp');
// 判断是否赋值
Session::has('name.item');
// 取值
Session::get('name.item');
// 删除
Session::delete('name.item');
```

其中 `set` 和 `delete` 方法只能支持二级数组，其他方法支持任意级数组操作。

## 助手函数

系统也提供了助手函数 `session` 完成相同的功能，例如：

```
// 赋值
session('name', 'thinkphp');
// 判断是否赋值
session('?name');
// 取值
session('name');
// 删除
```

```
session('name', null);
// 清除session
session(null);
```

## Request对象中读取Session

可以在Request对象中读取Session数据

```
public function index(Request $request) {
    // 读取某个session数据
    $request->session('user.name', '');
    // 获取全部session数据
    $request->session();
}
```

但 `Request` 类中不支持 `Session` 写入操作。

## 应用独立会话

> 多应用情况下默认 `Session` 是跨应用的，也就是说多应用之间是共享会话数据的，如果不希望共享会话数据，可以给每个应用设置不同的前缀 `prefix` 。

如果是File类型的话，默认的 `session` 会话数据保存在 `runtime/session` 目录下面，你可以设置 `path` 改变存储路径。

## Session驱动

默认的 `Session` 驱动采用文件缓存方式记录，并且支持如下配置

| 参数 | 描述 |
| --- | --- |
| path | session保存路径 |
| data_compress | 是否压缩数据 |
| gc_divisor | GC回收概率 |
| gc_probability | GC回收概率 |

除了文件类型之外，还可以支持直接使用缓存作为 `Session` 类型，例如：

```
return [
    'type'       => 'cache',
    'store'      => 'redis',
    'prefix'     => 'think',
]
```

表示使用 `redis` 作为 `session` 类型。

要以上的配置生效，请确保缓存配置文件 `cache.php` 中的 `stores` 中已经添加了 `redis` 缓存配置，例如：

```
return [
    'default'    =>    'file',
    'stores'    =>    [
        // 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',
        ],
    ],
];
```

## 自定义驱动

如果需要自定义 `Session` 驱动，你的驱动类必须实现 `think\contract\SessionHandlerInterface` 接口，包含了三个方法。

```
interface SessionHandlerInterface
{
    public function read(string $sessionId): string;
    public function delete(string $sessionId): bool;
    public function write(string $sessionId, string $data): bool;
}
```

`read` 方法是在调用 `Session::start()` 的时候执行，并且只会执行一次。

`write` 方法是在本地化会话数据的时候执行（调用 `Session::save()` 方法），系统会在每次请求结束的时候自动执行。

`delete` 方法是在销毁会话的时候执行（调用 `Session::destroy()` 方法）。

# 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，系统会在调用之前自动进行 `Cookie` 初始化工作。

支持的参数及默认值如下：

```
// 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'  => '',
```

## 基本操作

### 设置

```
// 设置Cookie 有效期为 3600秒
Cookie::set('name', 'value', 3600);
```

`Cookie` 数据不支持数组，如果需要请自行序列化后存入。

### 永久保存

```
// 永久保存Cookie
Cookie::forever('name', 'value');
```

### 删除

```
//删除cookie
Cookie::delete('name');
```

### 读取

```
// 读取某个cookie数据
Cookie::get('name');
// 获取全部cookie数据
Cookie::get();
```

## 助手函数

系统提供了 `cookie` 助手函数用于基本的 `cookie` 操作，例如：

```
// 设置
cookie('name', 'value', 3600);

// 获取
echo cookie('name');

// 删除
cookie('name', null);
```

# 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

默认系统会加载默认语言包，但如果需要多语言自动侦测及自动切换，你需要在全局的中间件定义文件中添加中间件定义：

```
'think\middleware\LoadLangPack',
```

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。多语言相关的设置在 `lang.php` 配置文件中进行设置。
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',
```

启用中间件后，系统会自动侦测和多语言自动切换，可以在配置文件设置自动侦测的多语言变量名：

```
// 自动侦测的GET变量名
'detect_var' => 'lang',
```

开启自动侦测后会会首先检查请求的URL或者Cookie中是否包含语言变量，然后根据 `HTTP_ACCEPT_LANGUAGE` 自动识别当前语言（并载入对应的语言包）。
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，可以配置：

```
// 设置允许的语言
'allow_lang_list'    =>    ['zh-cn', 'en-us']
```

如果希望使用 `Cookie` 保存语言，可以设置

```
// 使用Cookie保存
'use_cookie' => true,
// Cookie保存变量
'cookie_var' => 'think_lang',
```

设置后，自动检测的语言会通过 `Cookie` 记录下来，下次则直接通过 `Cookie` 判断语言。

## 语言变量定义

语言变量的定义，只需要在需要使用多语言的地方，写成：

```
Lang::get('add user error');
// 使用系统封装的助手函数
lang('add user error');
```

也就是说，字符串信息要改成 `Lang::get` 方法来表示。

语言定义一般采用英语来描述。

## 语言文件定义

自动加载的应用语言文件位于：

```
// 单应用模式
app\lang\当前语言.php
// 多应用模式
app\应用\lang\当前语言.php
```

如果你还需要加载其他的语言包，可以通过 `extend_list` 设置，例如：

```
'extend_list'    =>    [
    'zh-cn'    => [
        app()->getBasePath() . 'lang\zh-cn\app.php',
        app()->getBasePath() . 'lang\zh-cn\core.php',
    ],
]
```

> 目前核心框架仅内置 `zh-cn` 语言包，如果需要其它语言的提示，可以通过扩展语言包的方式自行加载。

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'])}
```

## 语言分组

首先你需要在lang.php配置文件中开启语言分组，

```
// 开启多语言分组
'allow_group'    =>    true
```

然后你可以在定义多语言的时候使用分组定义

```
return [
    'user'    =>    [
```

```
        'welcome'  => '欢迎回来',
        'login' => '用户登录',
        'logout' => '用户登出',
    ]
];
```

然后使用下面的方式获取多语言变量

```
Lang::get('user.login');
lang('user.login');
```

# 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安装额外的扩展。

假设表单代码如下：

```
<form action="/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');
    // 上传到本地服务器
    $savename = \think\facade\Filesystem::putFile( 'topic', $file);
}
```

`$file` 变量是一个 `\think\File` 对象，你可以获取相关的文件信息，支持使用 `SplFileObject` 类的属性和方法。

## 上传规则

默认情况下是上传到本地服务器，会在 `runtime/storage` 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：

```
runtime/storage/topic//20160510/42a79759f284b767dfcb2a0197904287.jpg
```

如果是多应用的话，上传根目录默认是 `runtime/index/storage` ，如果你希望上传的文件是可以直接访问或者下载的话，可以使用 `public` 存储方式。

```
$savename = \think\facade\Filesystem::disk('public')->putFile( 'topic', $file);
```

你可以在 `config/filesystem.php` 配置文件中配置上传根目录及上传规则，例如：

```
return [
```

```
    'default' =>  'local',
    'disks'   => [
        'local'  => [
            'type' => 'local',
            'root'   => app()->getRuntimePath() . 'storage',
        ],
        'public' => [
            'type'     => 'local',
            'root'       => app()->getRootPath() . 'public/storage',
            'url'        => '/storage',
            'visibility' => 'public',
        ],
        // 更多的磁盘配置信息
    ],
];
```

我们可以指定上传文件的命名规则，例如：

```
$savename = \think\facade\Filesystem::putFile( 'topic', $file, 'md5');
```

最终生成的文件名类似于：

```
runtime/storage/topic/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使用闭包方法，例如：

```
$savename = \think\facade\Filesystem::putFile( 'topic', $file,'uniqid');
```

## 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');
    $savename = [];
    foreach($files as $file){
        $savename[] = \think\facade\Filesystem::putFile( 'topic', $file);
    }
}
```

## 上传验证

支持使用验证类对上传文件的验证，包括文件大小、文件类型和后缀：

```
public function upload(){
    // 获取表单上传文件
    $files = request()->file();
    try {
        validate(['image'=>'filesize:10240|fileExt:jpg|image:200,200,jpg'])
            ->check($files);
        $savename = [];
        foreach($files as $file) {
            $savename[] = \think\facade\Filesystem::putFile( 'topic', $file);
        }
    } catch (\think\exception\ValidateException $e) {
        echo $e->getMessage();
    }
}
```

| 验证参数 | 说明 |
| --- | --- |
| fileSize | 上传文件的最大字节 |
| fileExt | 文件后缀，多个用逗号分割或者数组 |
| fileMime | 文件MIME类型，多个用逗号分割或者数组 |
| image | 验证图像文件的尺寸和类型 |

具体用法可以参考验证章节的内置规则-> 上传验证。

## 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');
// 获取上传文件的hash散列值
echo $file->md5();
echo $file->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');
// 获取上传文件的hash散列值
echo $file->hash('sha1');
echo $file->hash('md5');
```

# 命令行

ThinkPHP6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
version 6.0.0

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
  run               PHP Built-in Server for ThinkPHP
  version           show thinkphp framework version
 make
  make:command      Create a new command class
  make:controller   Create a new resource controller class
  make:event        Create a new event class
  make:listener     Create a new listener class
  make:middleware   Create a new middleware class
  make:model        Create a new model class
  make:service      Create a new Service class
  make:subscribe    Create a new subscribe class
  make:validate     Create a validate class
 optimize
  optimize:config   Build config and common file cache.
  optimize:facade   Build facade ide helper.
  optimize:route    Build app route cache.
  optimize:schema   Build database schema cache.
 route
  route:build       Build Annotation route rule.
  route:list        show route list.
```

```
  service
   service:discover  Discover Services for ThinkPHP
  vendor
   vendor:publish    Publish any publishable assets from vendor packages
```

`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 | 自动生成应用目录和文件 |
| help | 帮助 |
| list | 指令列表 |
| clear | 清除缓存指令 |
| run | 启动PHP内置服务器 |
| version | 查看当前框架版本号 |
| make:controller | 创建控制器类 |
| make:model | 创建模型类 |
| make:command | 创建指令类文件 |
| make:validate | 创建验证器类 |
| make:middleware | 创建中间件类 |
| make:event | 创建事件类 |
| make:listener | 创建事件监听器类 |
| make:subscribe | 创建事件订阅者类 |
| make:service | 创建系统服务类 |
| 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 |
| optimize:facade | 生成Facade注释 |
| route:build | 生成注解路由 |
| route:list | 查看路由定义 |
| service:discover | 自动注册扩展包的系统服务 |
| vendor:publish | 自动生成扩展的配置文件 |

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

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://0.0.0.0:8000/>
You can exit with `CTRL-C`
Document root is: D:\WWW\tp6/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。

支持制定IP和端口访问

```
>php think run -H tp.com -p 80
```

会显示

```
ThinkPHP Development server is started On <http://tp.com:80/>
You can exit with `CTRL-C`
Document root is: D:\WWW\tp6/public
```

然后你可以直接在浏览器里面访问

```
http://tp.com/
```

# 查看版本

## 查看版本

查看当前框架版本

```
php think version
```

输出显示

```
v6.0.0
```

# 自动生成应用目录

ThinkPHP 具备自动创建功能，可以用来自动生成需要的应用及目录结构和文件等。

> 本指令需要安装多应用扩展后才支持。

## 快速生成应用

如果使用了多应用模式，可以快速生成一个应用，例如生成 `demo` 应用的指令如下：

```
>php think build demo
```

如果看到输出

```
Successed
```

则表示自动生成应用成功。
会自动生成 `demo` 应用，自动生成的应用目录包含了 `controller` 、 `model` 和 `view` 目录以及 `common.php` 、 `middleware.php` 、 `event.php` 和 `provider.php` 等文件。
生成成功后，我们可以直接访问 `demo` 应用
会显示

```
您好！这是一个[demo]示例应用
```

## 应用结构自定义

如果你希望自定义生成应用的结构，可以在app目录下增加一个 `build.php` 文件，内容如下：

```php
return [
    // 需要自动创建的文件
    '__file__'   => [],
    // 需要自动创建的目录
    '__dir__'    => ['controller', 'model', 'view'],
    // 需要自动创建的控制器
    'controller' => ['Index'],
    // 需要自动创建的模型
    'model'      => ['User'],
    // 需要自动创建的模板
    'view'       => ['index/index'],
];
```

可以给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（默认会生成 `common.php` 、 `middleware.php` 、 `event.php` 和 `provider.php` 文件，无需定义）
- `controller` 表示生成控制器类
- `model` 表示生成模型类
- `view` 表示生成模板文件（支持子目录）

并且会自动生成应用的默认 `Index` 访问控制器文件用于显示应用的欢迎页面。

# 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` 应用的 `Blog` 控制器类库文件

```
>php think make:controller index@Blog
```

如果是单应用模式，则无需传入应用名

```
>php think make:controller 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
{
    /**
     * 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;


class Blog
{
    //
}
```

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\controller\test\Blog` 控制器类。

可以支持 --api 参数生成用于API接口的资源控制器。

## 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` 应用的 `Blog` 模型类库文件

```
>php think make:model index@Blog
```

如果是单应用模式，无需传入应用名

```
>php think make:model 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 BlogModel
```

## 快速生成中间件

可以使用下面的指令生成一个中间件类。

```
>php think make:middleware Auth
```

会自动生成一个 `app\middleware\Auth` 类文件。

## 创建验证器类

可以使用

```
>php think make:validate index@User
```

生成一个 `app\index\validate\User` 验证器类，然后添加自己的验证规则和错误信息。

# 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。
如果不需要保留空目录，可以使用

```
php think clear --dir
```

清除日志目录

```
php think clear --log
```

清除日志目录并删除空目录

```
php think clear --log --dir
```

清除数据缓存目录

```
php think clear --cache
```

清除数据缓存目录并删除空目录

```
php think clear --cache --dir
```

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\tp\runtime\log\
```

# 生成数据表字段缓存

## 生成数据表字段缓存 `optimize:schema`

> 字段缓存仅在部署模式下生效，并且仅适用于使用 `think-orm` 的情况，如果你使用了其它的ORM库，则不支持生成。

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
```

如果是多应用模式，你可以使用下面的指令生成 `admin` 应用的字段缓存。

```
php think optimize:schema admin
```

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。

> 没有继承think\Model类的（抽象）模型类不会生成。如果定义了公共模型类，最好把公共模型类定义为抽象类（ `abstract` ）。

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。
如果是多应用模式的话，需要增加应用名参数调用指令

```
php think optimize:route index
```

# 输出路由定义

## 输出并生成路由列表

假设你的路由定义文件内容为：

```
Route::get('think', function () {
    return 'hello,ThinkPHP6!';
});

Route::resource('blog', 'Blog');

Route::get('hello/:name', 'index/hello')->ext('html');
```

可以使用下面的指令查看定义的路由列表

```
php think route:list
```

如果是多应用模式的话，需要改成

```
php think route:list index
```

输出结果类似于下面的显示：

```
+----------------+-------------+--------+-------------+
| Rule           | Route       | Method | Name        |
+----------------+-------------+--------+-------------+
| think          | <Closure>   | get    |             |
| hello/<name>   | index/hello | get    | index/hello |
| blog           | Blog/index  | get    | Blog/index  |
| blog           | Blog/save   | post   | Blog/save   |
| blog/create    | Blog/create | get    | Blog/create |
| blog/<id>/edit | Blog/edit   | get    | Blog/edit   |
| blog/<id>      | Blog/read   | get    | Blog/read   |
| blog/<id>      | Blog/update | put    | Blog/update |
| blog/<id>      | Blog/delete | delete | Blog/delete |
+----------------+-------------+--------+-------------+
```

并且同时会在runtime目录下面生成一个 `route_list.php` 的文件，内容和上面的输出结果一致，方便你随时查看。

> 如果你的路由定义发生改变的话， 则需要重新调用该指令，会自动更新上面生成的缓存文件。

## 输出样式

支持定义不同的样式输出，例如：

```
php think route:list box
```

输出结果变为：

```
┌──────────────────┬──────────────┬────────┬──────────────┐
| Rule             | Route        | Method | Name         |
├──────────────────┼──────────────┼────────┼──────────────┤
| think            | <Closure>    | get    |              |
| hello/<name>     | index/hello  | get    | index/hello  |
| blog             | Blog/index   | get    | Blog/index   |
| blog             | Blog/save    | post   | Blog/save    |
| blog/create      | Blog/create  | get    | Blog/create  |
| blog/<id>/edit   | Blog/edit    | get    | Blog/edit    |
| blog/<id>        | Blog/read    | get    | Blog/read    |
| blog/<id>        | Blog/update  | put    | Blog/update  |
| blog/<id>        | Blog/delete  | delete | Blog/delete  |
└──────────────────┴──────────────┴────────┴──────────────┘
```

```
php think route:list box-double
```

输出结果变为：

```
╔══════════════════╤══════════════╤════════╤══════════════╗
╟ Rule             | Route        | Method | Name         ╢
╠══════════════════╪══════════════╪════════╪══════════════╣
╟ think            | <Closure>    | get    |              ╢
╟ hello/<name>     | index/hello  | get    | index/hello  ╢
╟ blog             | Blog/index   | get    | Blog/index   ╢
╟ blog             | Blog/save    | post   | Blog/save    ╢
╟ blog/create      | Blog/create  | get    | Blog/create  ╢
╟ blog/<id>/edit   | Blog/edit    | get    | Blog/edit    ╢
╟ blog/<id>        | Blog/read    | get    | Blog/read    ╢
╟ blog/<id>        | Blog/update  | put    | Blog/update  ╢
╟ blog/<id>        | Blog/delete  | delete | Blog/delete  ╢
╚══════════════════╧══════════════╧════════╧══════════════╝
```

```
php think route:list markdown
```

输出结果变为：

```
| Rule            | Route       | Method | Name        |
|-----------------|-------------|--------|-------------|
| think           | <Closure>   | get    |             |
| hello/<name>    | index/hello | get    | index/hello |
| blog            | Blog/index  | get    | Blog/index  |
| blog            | Blog/save   | post   | Blog/save   |
| blog/create     | Blog/create | get    | Blog/create |
| blog/<id>/edit  | Blog/edit   | get    | Blog/edit   |
| blog/<id>       | Blog/read   | get    | Blog/read   |
| blog/<id>       | Blog/update | put    | Blog/update |
| blog/<id>       | Blog/delete | delete | Blog/delete |
```

## 排序支持

如果你希望生成的路由列表按照路由规则排序，可以使用

```
php think route:list -s rule
```

输出结果变成：

```
+-----------------+-------------+--------+-------------+
| Rule            | Route       | Method | Name        |
+-----------------+-------------+--------+-------------+
| blog            | Blog/index  | get    | Blog/index  |
| blog            | Blog/save   | post   | Blog/save   |
| blog/<id>       | Blog/read   | get    | Blog/read   |
| blog/<id>       | Blog/update | put    | Blog/update |
| blog/<id>       | Blog/delete | delete | Blog/delete |
| blog/<id>/edit  | Blog/edit   | get    | Blog/edit   |
| blog/create     | Blog/create | get    | Blog/create |
| hello/<name>    | index/hello | get    | index/hello |
| think           | <Closure>   | get    |             |
+-----------------+-------------+--------+-------------+
```

同样的，你还可以按照请求类型排序

```
php think route:list -s method
```

输出结果变为：

```
+-----------------+-------------+--------+-------------+
| Rule            | Route       | Method | Name        |
+-----------------+-------------+--------+-------------+
| blog/<id>       | Blog/delete | delete | Blog/delete |
```

```
| think           | <Closure>   | get    |             |
| hello/<name>    | index/hello | get    | index/hello |
| blog            | Blog/index  | get    | Blog/index  |
| blog/create     | Blog/create | get    | Blog/create |
| blog/<id>/edit  | Blog/edit   | get    | Blog/edit   |
| blog/<id>       | Blog/read   | get    | Blog/read   |
| blog            | Blog/save   | post   | Blog/save   |
| blog/<id>       | Blog/update | put    | Blog/update |
+-----------------+-------------+--------+-------------+
```

> 支持排序的字段名包括： `rule` 、 `route` 、 `name` 、 `method` 和 `domain` （全部小写）。

## 输出详细信息

如果你希望看到更多的路由参数和变量规则，可以使用

```
php think route:list -m
```

输出结果变为：

```
+----------------+-------------+--------+-------------+--------+----------------
---------+---------+
| Rule           | Route       | Method | Name        | Domain | Option
         | Pattern |
+----------------+-------------+--------+-------------+--------+----------------
---------+---------+
| think          | <Closure>   | get    |             |        | []
         | []      |
| hello/<name>   | index/hello | get    | index/hello |        | {"ext":"html"}
         | []      |
| blog           | Blog/index  | get    | Blog/index  |        | {"complete_mat
ch":true} | []      |
| blog           | Blog/save   | post   | Blog/save   |        | {"complete_mat
ch":true} | []      |
| blog/create    | Blog/create | get    | Blog/create |        | []
         | []      |
| blog/<id>/edit | Blog/edit   | get    | Blog/edit   |        | []
         | []      |
| blog/<id>      | Blog/read   | get    | Blog/read   |        | []
         | []      |
| blog/<id>      | Blog/update | put    | Blog/update |        | []
         | []      |
| blog/<id>      | Blog/delete | delete | Blog/delete |        | []
         | []      |
+----------------+-------------+--------+-------------+--------+----------------
---------+---------+
```

# 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，运行指令

```
php think make:command Hello hello
```

会生成一个 `app\command\Hello` 命令行指令类，我们修改内容如下：

```
<?php
namespace app\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 `config/console.php` 文件

```
<?php
return [
    'commands' => [
        'hello' => 'app\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 classma
ps 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 kancloud
```

输出

```
Hello kancloud!
```

添加 `city` 选项

```
php think hello kancloud --city shanghai
```

输出

```
Hello kancloud!
From shanghai
```

> 注意看参数和选项的调用区别

如果需要生成一个指定的命名空间，可以使用：

```
php think make:command app\index\Command second
```

## 在控制器中调用命令

支持在控制器的操作方法中直接调用命令，例如：

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

use think\facade\Console;
```

```
class Index
{
    public function hello($name)
    {
        $output = Console::call('hello', [$name]);

        return $output->fetch();
    }
}
```

访问该操作方法后，例如：

```
http://serverName/index/hello/name/thinkphp
```

页面会输出

```
Hello thinkphp!
```

# 扩展库

# 数据库迁移工具

## 数据库迁移工具

使用数据库迁移工具可以将数据库结构和数据很容易的在不同的数据库之间管理迁移。
在以前，为了实现“程序安装”，你可能会导出一份sql文件，安装时，用程序解析这个sql文件，执行里面的语句，这样做有诸多的局限性，但现在使用数据库迁移工具，你可使用一个强大的类库API来创建数据库结构和记录，并且可以容易的安装到Mysql，sqlite，sqlserver等数据库。
原项目手册：phinx
使用范例：

> 使用之前你应当正确的连接到数据库,不论是mysql,sqlite,sqlserver

### 安装

```
composer require topthink/think-migration
```

### 创建迁移工具文件

```
//执行命令,创建一个操作文件,一定要用大驼峰写法,如下
php think migrate:create AnyClassNameYouWant
//执行完成后,会在项目根目录多一个database目录,这里面存放类库操作文件
//文件名类似/database/migrations/20190615151716_any_class_name_you_want.php
```

### 编辑文件

```
<?php

use think\migration\Migrator;
use think\migration\db\Column;

class  AnyClassNameYouWant extends  Migrator
{
    /**
    * Change Method.
    *
    * Write your reversible migrations using this method.
    *
    * More information on writing migrations is available here:
    * http://docs.phinx.org/en/latest/migrations.html#the-abstractmigration-cla
ss
    *
    * The following commands can be used in this method and Phinx will
```

```
     * automatically reverse them when rolling back:
     *
     * createTable
     * renameTable
     * addColumn
     * renameColumn
     * addIndex
     * addForeignKey
     *
     * Remember to call "create()" or "update()" and NOT "save()" when working
     * with the Table class.
     */

    public  function  change()
    {
        // create the table
        $table  =  $this->table('users',array('engine'=>'MyISAM'));
        $table->addColumn('username', 'string',array('limit'  =>  15,'default'=>
'','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,'default'=>0,
'comment'=>'登陆状态'))
        ->addColumn('login_code', 'string',array('limit'  =>  32,'default'=>0,'
comment'=>'排他性登陆标识'))
        ->addColumn('last_login_ip', 'integer',array('limit'  =>  11,'default'=>
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,'default'=>0,'c
omment'=>'删除状态，1已删除'))
        ->addIndex(array('username'), array('unique'  =>  true))
        ->create();
    }

}
```

## 执行迁移工具

```
php think migrate:run
//此时数据库便创建了prefix_users表.
```

> 数据库会有一个migrations表,这个是工具使用的表,不要修改

## 例子

```
 $table = $this->table('followers', ['id' => false, 'primary_key' => ['user_id',
```

```
  'follower_id']]);
$table->addColumn('user_id', 'integer')
      ->addColumn('follower_id', 'integer')
      ->addColumn('created', 'datetime')
      ->addIndex(['email','username'], ['limit' => ['email' => 5, 'username' =>
2]])
      ->addIndex('user_guid', ['limit' => 6])

     ->create();
```

表支持的参数选项

| 选项 | 描述 |
| --- | --- |
| comment | 给表结构设置文本注释 |
| row_format | 设置行记录模格式 |
| engine | 表引擎 (默认 `InnoDB` ) |
| collation | 表字符集 (默认 `utf8\_general\_ci` ) |
| signed | 是否无符号 `signed` (默认 `true` ) |

常用列

- biginteger
- binary
- boolean
- date
- datetime
- decimal
- float
- integer
- string
- text
- time
- timestamp
- uuid

所有的类型都支持的参数

| Option | Description |
| --- | --- |

| | |
|---|---|
| limit | 文本或者整型的长度 |
| length | `limit` 别名 |
| default | 默认值 |
| null | 允许 `NULL` 值 (不该给主键设置 |
| after | 在哪个字段名后 (只对MySQL有效) |
| comment | 给列设置文本注释 |

索引的用法

```
      ->addIndex(['email','username'], ['limit' => ['email' => 5, 'username' => 2]])
      ->addIndex('user_guid', ['limit' => 6])
      ->addIndex('email',['type'=>'fulltext'])
```

如上面例子所示，默认是 `普通索引` ，mysql可设置生效 `复合索引` ，mysql可以设置 `fulltext` .

## 自动版本升级降级

该项目可以升级和还原，就像git/svn一样rollback。

如果希望实现自动升级降级，那就把逻辑写在change方法里，只最终调用 `create` 和 `update` 方法，不要调用 `save` 方法。

`change` 方法内仅支持以下操作

- createTable
- renameTable
- addColumn
- renameColumn
- addIndex
- addForeignKey

如果真的有调用其他方法，可以写到 `up` 和 `down` 方法里，这里的逻辑不支持自动还原，up写升级的逻辑，down写降级的逻辑。

```
    public function change()
    {
        // create the table
        $table = $this->table('user_logins');
        $table->addColumn('user_id', 'integer')
              ->addColumn('created', 'datetime')
              ->create();
```

```
    }

    /**
     * Migrate Up.
     */
    public function up()
    {

    }

    /**
     * Migrate Down.
     */
    public function down()
    {

    }
```

# 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。

## 安装

首先通过 composer 安装

```
composer require topthink/think-worker
```

## 使用

### 使用 `Workerman` 作为 `HttpServer`

在命令行启动服务端

```
php think worker
```

然后就可以通过浏览器直接访问当前应用

```
http://localhost:2346
```

linux下面可以支持下面指令

```
php think worker [start|stop|reload|restart|status]
```

`workerman` 的参数可以在应用配置目录下的 `worker.php` 里面配置。

> 由于 `onWorkerStart` 运行的时候没有 `HTTP_HOST` ，因此最好在应用配置文件中设置 `app_host`

### SocketServer

在命令行启动服务端（需要 `2.0.5+` 版本）

```
php think worker:server
```

默认会在0.0.0.0:2345开启一个 `websocket` 服务。

如果需要自定义参数，可以在 `config/worker_server.php` 中进行配置，包括：

| 配置参数 | 描述 |
| --- | --- |
| protocol | 协议 |
| host | 监听地址 |
| port | 监听端口 |
| socket | 完整的socket地址 |

并且支持 `workerman` 所有的参数（包括全局静态参数）。

也支持使用闭包方式定义相关事件回调。

```
return [
    'socket'    =>  'http://127.0.0.1:8000',
    'name'      =>  'thinkphp',
    'count'     =>  4,
    'onMessage' =>  function($connection, $data) {
        $connection->send(json_encode($data));
    },
];
```

也支持使用自定义类作为 `Worker` 服务入口文件类。例如，我们可以创建一个服务类（必须要继承 `think\worker\Server` ），然后设置属性和添加回调方法

```
<?php
namespace app\http;

use think\worker\Server;

class Worker extends Server
{
    protected $socket = 'http://0.0.0.0:2346';

    public function onMessage($connection,$data)
    {
        $connection->send(json_encode($data));
    }
}
```

支持 `workerman` 所有的回调方法定义（回调方法必须是public类型）

然后在 `worker_server.php` 中增加配置参数：

```
return [
    'worker_class'  =>  'app\http\Worker',
];
```

定义该参数后，其它配置参数均不再有效。

在命令行启动服务端

```
php think worker:server
```

然后在浏览器里面访问

```
http://localhost:2346
```

如果在Linux下面，同样支持reload|restart|stop|status 操作

```
php think worker:server reload
```

> 注意:使用默认worker做服务器时, `dump` 会打印到命令行.

## 关于上传文件

当按照默认的worker做http服务器时,并不能直接使用 `request()->file('image')` 来获得上传的文件,具体可以参考workerman的上传文件第6点.因此只能迂回的使用 `Filesystem` .无论怎样,不影响其 `getMime()` 等方法的正确性.

```
// $file = request()->file('image');
$file_data  =  $_FILES[0]['file_data'];
//$tmp_file  = tempnam('','tm_');  这种写法最终保存时扩展名为.tmp
$tmp_file  = sys_get_temp_dir().'/'.uniqid().'.'.explode('/',$_FILES[0]['file_t
ype'])[1];
file_put_contents($tmp_file,$file);
$file  =  new  File($tmp_file);
$savename  =  Filesystem::putFile('upload',$file);
echo  $savename;
```

# think助手工具库

## 常用的一些扩展类库

目前已有字符串操作,数组操作.

安装

```
composer require topthink/think-helper
```

> 更新完善中

> 以下类库都在 `\think\helper` 命名空间下

## Str

> 字符串操作

```
use think\\helper\\Str;

// 检查字符串中是否包含某些字符串
Str::contains($haystack, $needles)

// 检查字符串是否以某些字符串结尾
Str::endsWith($haystack, $needles)

// 获取指定长度的随机字母数字组合的字符串
Str::random($length = 16)

// 字符串转小写
Str::lower($value)

// 字符串转大写
Str::upper($value)

// 获取字符串的长度
Str::length($value)

// 截取字符串
Str::substr($string, $start, $length = null)

//驼峰转下划线
Str::snake($value, $delimiter  =  '_')

//下划线转驼峰(首字母小写)
```

```
Str::camel($value)

//下划线转驼峰(首字母大写)
Str::studly

//转为首字母大写的标题格式
Str::title($value)
```

## Arr

> 数组操作

```
use think\\helper\\Arr;
```

### 判断能否当做数组一样访问

```
//数组返回真
//模型查询的结果也为真
Arr::accessible($value)
```

### 向数组添加一个元素,支持"点"分割

```
Arr::add($array, $key, $value)
//如下操作
$arr  = [];
$arr = Arr::add($arr,'name.3.ss','thinkphp'); //本行结果$arr['name'][3]['ss'] =
'thinkphp'
Arr::add($arr,'name','thinkphp2');//本行不会产生影响,因为'name'已存在.
```

### 将数据集管理类转换为数组

```
Arr::collapse($array)
```

### 排列数组组合

```
$arr  =  Arr::crossJoin(['dd'],['ff'=>'gg'],[2],[['a'=>'mm','kk'],'5']);
//上面行没有什么意义,但是可以看到该函数的作用,数组索引被忽略,数组得值被全部组合
$arr  =  Arr::crossJoin(['a','b','c'],['aa','bb','cc','dd']);
//这一行可以看到组合的效果,返回一个二维数组,第二维每个数组是的给定值排列,比如(a,aa),(a,bb),
(a,cc)...直到(c,dd)
```

# 验证码

## 安装

首先使用 `Composer` 安装 `think-captcha` 扩展包：

```
composer require topthink/think-captcha
```

> 验证码库需要开启Session才能生效。

## 使用

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
};
```

如果是多应用模式下，你需要自己注册一个验证码的路由。

```
Route::get('captcha/[:config]','\\think\\captcha\\CaptchaController@index');
```

## 配置

`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 |
| math | 使用算术验证码 | false |
| 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 |

直接在应用的 `config` 目录下面的 `captcha.php` 文件中进行设置即可，例如下面的配置参数用于输出4位数字验证码。

```
return [
    'length'    =>  4,
    'codeSet'   =>  '0123456789',
];
```

## 自定义验证码

如果需要自己独立生成验证码，可以调用 `Captcha` 类（ `think\captcha\facade\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\facade\Captcha;

class Index
{
    public function verify()
    {
        return Captcha::create();
    }
}
```

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>
```

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
}
```

如果你需要生成多个不同设置的验证码，可以使用下面的配置方式：

```
<?php
return [
    'verify'=>[
        'codeSet'=>'1234567890'
    ]
];
```

使用指定的配置生成验证码:

```
return Captcha::create('verify');
```

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：

修改或新建配置文件如下:

```
<?php
return [
    'verify'=>[
        'fontttf'=>'1.ttf'
    ]
];
```

```
return Captcha::create('verify');
```

> 默认的验证码字符已经剔除了易混淆的 `1l0o` 等字符

# Swoole

本篇内容主要讲述了最新的 `think-swoole` 扩展的使用。目前仅支持Linux环境或者MacOs下运行，要求 `swoole` 版本为 `4.3.1+` 。

> 由于 `think-swoole` 是基于 `swoole` 的，要了解这个扩展如何使用，首先需要对 `swoole` 有一定的了解，这也是本文阅读的前提，具体可以参考 Swoole官方文档内容：https://wiki.swoole.com

## 安装

首先按照 `Swoole` 官网说明安装 `swoole` 扩展，然后使用

```
composer require topthink/think-swoole
```

安装 `think-swoole` 扩展。

> 由于 `Swoole` 不支持 `windows` 环境，所以你无法在 `windows` 环境下测试，只能使用虚拟机或者 `WSL` 环境测试。

## `HTTP` 服务

直接在命令行下启动HTTP服务端。

```
php think swoole
```

启动完成后，默认会在 `0.0.0.0:80` 启动一个HTTP Server，可以直接访问当前的应用。相关配置参数可以在 `config/swoole.php` 里面配置（具体参考配置文件内容）。
支持的其它操作包括：
启动HTTP服务（默认）

```
php think swoole start
```

停止服务

```
php think swoole stop
```

重启服务

```
php think swoole restart
```

`reload` 服务

```
php think swoole reload
```

### 守护进程模式

如果需要使用守护进程方式运行，可以配置

```
'options'   =>  [
    'daemonize' =>  true
]
```

## 热更新

由于 `Swoole` 服务运行过程中PHP文件是常驻内存运行的，这样可以避免重复读取磁盘、重复解释编译PHP，以便达到最高性能。所以更改业务代码后必须手动 `reload` 或者 `restart` 才能生效。

`think-swoole` 扩展提供了热更新功能，在检测到相关目录的文件有更新后会自动 `reload` ，从而不需要手动进行 `reload` 操作，方便开发调试。

如果你的应用开启了调试模式，默认是开启热更新的。原则上，在部署模式下不建议开启文件监控，一方面有性能损耗，另外一方面对文件所做的任何修改都需要确认无误才能进行更新部署。

热更新的默认配置如下：

```
'hot_update' => [
    'enable'  => env( name: 'APP_DEBUG', default: false),
    'name'    => ['*.php'],
    'include' => [app_path()],
    'exclude' => [],
],
```

当我们在应用的根目录下定义一个特殊的 `.env` 环境变量文件，里面设置了 `APP_DEBUG = true` 会默认开启热更新，你也可以直接把 `enable` 设置为true。

参数说明：

| 参数 | 说明 |
| --- | --- |
| enable | 是否开启热更新 |
| name | 简单点说就是监控那些类型的文件变动 |
| | 简单点说就是监控那些路径下的文件变 |

| | |
|---|---|
| | 动 |
| exclude | 排除目录 |

## 连接池

`think-swoole` 默认有实现数据库和缓存连接池功能，涵盖了日常开发的主要场景。

最新的 `swoole` 版本支持一键协程，比如 `redis` 、 `mysql` 等等，很方便。连接池是在这个基础上，解决一些问题和对性能的再一次提升。

要开启一键协程，需要配置如下参数

```
'coroutine'   => [
    'enable' => true,
    'flags'  => SWOOLE_HOOK_ALL,
],
```

这里需要设置为true，默认已经打开，flags默认即可。

连接池的配置参数如下：

```
//连接池
'pool' => [
    'db' => [
        'enable' => true,
        'max_active' => 3,
        'max_wait_time' => 5,
    ],
    'cache' => [
        'enable' => true,
        'max_active' => 3,
        'max_wait_time' => 5,
    ],
],
```

参数说明：

| 参数 | 说明 |
|---|---|

| 参数 | 说明 |
| --- | --- |
| enable | 开关，不需要设置false |
| max_active | 最大连接数，超过将不再新建连接 |
| max_wait_time | 超时时间 |

其中的 `max_active` 和 `max_wait_time` 需要根据自身业务和环境进行适当调整，最大化提高系统负载。

# 附录

# 助手函数

## 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 |
| app | 快速获取容器中的实例 支持依赖注入 |
| bind | 快速绑定对象实例 |
| cache | 缓存管理 |
| 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 |
| cookie | Cookie管理 |
| download | 获取\think\response\Download对象实例 |
| dump | 浏览器友好的变量输出 |
| env | 获取环境变量 |
| event | 触发事件 |
| 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 |
| invoke | 调用反射执行callable 支持依赖注入 |
| json | JSON数据输出 |
| jsonp | JSONP数据输出 |
| lang | 获取语言变量值 |
| parse_name | 字符串命名风格转换 |
| redirect | 重定向输出 |
| request | 获取当前Request对象 |
| response | 实例化Response对象 |
| 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 |
| display | 渲染内容输出 |
| xml | XML数据输出 |
| app_path | 当前应用目录 |
| base_path | 应用基础目录 |
| config_path | 应用配置目录 |
| public_path | web根目录 |
| root_path | 应用根目录 |
| runtime_path | 应用运行时目录 |

> 可以在应用的公共函数文件中重写上面这些助手函数。

# 升级指导

`V6.0` 版本保持兼容升级，但不兼容 `5.*` 版本的升级。

## `5.1` 升级到 `6.0` 版本

不建议老的项目升级到新版，除非你有重构计划，否则就算升级了也只是表面上升级了。

> 本文主要用于指导开发者从 `5.1` 升级到 `6.0` 最新（RC）版本，由于 `6.0` 不支持 `5.1` 的无缝升级，下面的升级步骤和指导仅供学习参考，或有遗漏及考虑不周之处，因此不保证你的所有功能都能正常升级。

`6.0` 版本必须使用 `composer` 安装，所以你需要首先安装新的 `6.0` 版本，然后把原来 `5.1` 的文件复制进去，完成升级工作。

```
composer create-project topthink/think:6.0.0 tp
```

安装完成后，把原来 `application` 目录下面的文件复制到 `app` 目录下面，然后把 `config` 和 `route` 目录下的文件复制到同名目录下。接下来，按照下面的步骤来升级操作。
从 `5.1` 多模块迁移到 `6.0` 的多应用后，应用类库的命名空间原则上可以无需调整，但不再支持跨应用调用（包括路由及模板，每个应用的路由是独立的），这点务必引起重视。如果你的应用根命名空间不是默认的 `app` 需要改成 `app` 。

- 第一步：应用配置
- 第二步：配置调整
- 第三步：路由和请求调整
- 第四步：控制器和视图调整
- 第五步：数据库和模型调整
- 第六步：行为调整
- 第七步：其它调整及注意事项

### 第一步：应用配置

如果原来你使用了多模块开发模式，直接改成新版的多应用模式是最简单的，需要额外安装多应用模式扩展。

```
composer require topthink/think-multi-app
```

如果你自定义了访问控制器的名称，需要修改 `route.php` 配置文件中的 `controller_layer` 值。

```
// 访问控制器层名称
'controller_layer'      => 'controller',
```

如果你开启了控制器类的后缀，需要设置 `route.php` 配置文件中的 `controller_suffix` 值。

```
// 开启控制器后缀
'controller_suffix'     => true,
```

如果自定义了空控制器的名称，则需要设置 `route.php` 配置文件中的 `empty_controller` 值。

```
// 空控制器名
'empty_controller'      => 'Error',
```

## 第二步：配置调整

请按照如下顺序检查及调整你的配置文件和相关配置代码。

### 应用的配置文件

如果是多应用的话，应用配置文件应当放入应用下的 `config` 目录。全局配置文件位置无需调整。

### 配置获取调整

原来获取一级配置参数的方式

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

需要改成

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

所有的配置读取必须从第一级配置开始，例如，原来的

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

必须改成

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

### 废弃动态设置

动态更改配置参数的用法已经废弃，下面的用法不再支持。

```
Config::set('route.default_return_type', 'json');
```

如果你需要把数据库的配置参数读入配置，可以使用

```
$config = Db::name('config')->column('value', 'name');
Config::set($config, 'route');
```

### `Config` 类不再支持数组方式读取

`Config` 类不再使用 `ArrayAccess` 接口，因此不再支持数组方式读取。

### 路由和URL配置独立

路由和URL请求相关的配置参数独立为 `route.php` 配置文件，而不再使用 `app.php` 配置文件。

## 第三步：路由和请求调整

### 路由定义文件位置调整

单应用模式下，路由定义文件和之前一样就在 `route` 目录下面，如果你的项目是采用了多应用模式的话，每个应用的路由定义和匹配都是独立的，也没有模块的概念，路由定义文件的位置应该是在 `应用/route` 下面，例如：

```
app/index/route/route.php //  index应用的路由定义文件
app/index/route/web.php //  index应用的第二个路由定义文件
app/admin/route/route.php // admin应用的路由定义文件
```

> 应用的路由规则其实是定义的入口文件（或者应用名）后面的URL部分，而不包含应用。

### 路由注册方法调整

首先如果你的路由定义采用的是返回数组形式，全部改成方法定义。

例如：

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

必须改成：

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

如果路由定义方法（包括 `rule` / `get` / `post` / `put` / `delete` / `patch` / `miss` / `group` 等方法）使用了 `option` 和 `pattern` 参数，全部改成方法调用形式，例如原来的：

```
Route::get('hello/:name', 'index/hello', [ 'ext'  =>  'html'], [ 'name'  => '\w+']);
```

需要改成

```
Route::get('hello/:name', 'index/hello')
    ->ext('html')
    ->pattern([ 'name'  => '\w+']);
```

## 路由分组调整

如果路由分组定义使用了数组，改成闭包方式定义，例如：

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

必须改成

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

如果你需要注册一个虚拟的路由分组，可以直接在第一个参数使用闭包

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

## 取消了 `url_controller_layer` 配置

改为在 `route.php` 配置文件中使用 `controller_layer` 设置。

## 取消 `controller_suffix` 配置

改为在 `route.php` 配置文件中使用 `controller_suffix` 设置。
同时 `class_suffix` 配置参数已经无效。

### 取消 `mergeExtraVars` 方法和对应参数

改为在路由规则中明确指定变量规则。

### `allowCrossDomain` 方法参数调整

取消原来的第一个参数。

### `header` 方法取消

需要单独改变Header信息的直接通过中间件统一处理。

### 取消 `Request` 类的 `hook` 方法

该方法已经在最新版本中取消。如果你使用了该功能，在自定义请求对象 `app\Request` 中直接增加相应的方法即可。并确保 `provider.php` 文件中添加如下绑定：

```
'think\Request' => \app\Request::class,
```

### 取消 `URL` 参数模式配置

原来的URL参数模式配置参数 `url_param_type` ，统一使用参数/值的方式。如果你设置了该配置参数为1，必须改成定义路由的方式。

### 取消别名路由

因为使用场景有限和性能开销问题，取消原来的别名路由功能，建议使用资源路由或者单独的路由替代。

### 取消快捷路由

因为使用场景有限和不太符合规范，取消了原来的控制器快捷路由功能。

### 取消空操作功能

建议使用分组MISS路由功能或者控制器的 `__call` 方法替代。

## 第四步：控制器和视图调整

### `think\Controller` 类取消

系统不再提供基础控制器类 `think\Controller` ，原来的 `success` 、 `error` 、 `redirect` 和 `result` 方法需要自己在基础控制器类里面实现。
系统默认在应用目录下面提供了一个 `app\BaseController` 基础类，或者你可以直接放入你的应用里面，继

承使用。

你可以安装下面的扩展用于支持旧版本的跳转操作

```
composer require liliuwei/thinkphp-jump
```

### 视图和模板引擎从核心分离

模板引擎类不再内置到核心框架，但使用

```
composer create-project topthink/think
```

会默认安装该组件（如果不需要使用的话可以自己卸载 `topthink/think-view` ）。
安装后，由于内置的 `think\Controller` 类已经取消，如果你的控制器类需要调用 `fetch` / `display` / `assign` 等视图方法，必须改为调用 `think\facade\View` 类，如果是使用 `view` 助手函数方式的话，可以无需调整。

```
View::assign('name', $name);
View::fetch();
```

### `share` 方法取消

原来视图类的 `share` 方法取消，可以使用

```
think\facade\View::assign($vars);
```

## 第五步：数据库和模型调整

### `Db` 改为使用门面对象

新版的 `Db` 类不再是静态类，需要使用 `think\facade\Db` 门面进行静态代理。

```
\think\facade\Db::name('user')->find();
```

### 数据库配置信息调整

数据库配置文件或者 `connect` 方法取消 `DSN` 数据库配置定义方式，全部采用数组方式配置定义。

```
Db::connect('mysql://root:1234@127.0.0.1:3306/thinkphp#utf8')
    ->table('user')
    ->find();
```

必须改成

```
Db::connect('db_config')
    ->table('user')
    ->find();
```

并且按照新版的规范在数据库配置文件中增加 `db_config` 连接信息。

## 取消 `fetchPdo` 方法

取消了 `Query` 类的 `fetchPdo` 方法，需要的时候直接使用 `getPdo` 方法替代。

## 取消查询方法传入 `Query` 对象

取消Query类的CURD查询方法传入当前对象，如果需要请使用闭包替代。

## `insert` / `insertGetId` / `insertAll` 方法取消 `replace` 参数

`insert` / `insertGetId` / `insertAll` 方法的第二个 `replace` 参数已经取消，改为使用 `replace` 方法。

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->insert($data, true);
```

需要改为

```
$data = ['foo' => 'bar', 'bar' => 'foo'];
Db::name('user')->replace()->insert($data);
```

## 取消 `db` 和 `model` 助手函数

这两个助手函数 `5.1` 版本已经不再建议使用了， `6.0` 版本已经废弃掉这两个助手函数，请直接使用 `\think\facade\Db` 类静态方法和实际的模型类调用。

## 取消 `setInc` / `setDec` 方法

取消Query类的 `setInc` / `setDec` 方法，统一使用 `inc` / `dec` 方法替代。例如：

```
Db::name('user')->where('id', 1)
    ->inc('exp')
    ->dec('score')
    ->update();
```

## 取消 `join` 方法的批量操作

`join` 方法不再支持批量操作多个表，如果你使用了 `join` 方法批量操作，需要改成每个表单独调用一次 `join` 方法。

### 取消 `setField` 方法

取消Query类的 `setField` 方法，请直接使用 `data` 方法或者 `update` 方法。

### 取消 `__TABLE_NAME__` 支持

`table` 方法取消 `__TABLE_NAME__` 支持，必须明确调用完整表名或者使用 `name` 方法。

### 取消 `whereOr` 等方法传入 `Query` 对象

因为 `Query` 对象查询只能使用一次，除了 `where` 方法本身可以传入 `Query` 对象外，其它的所有 `where` 查询方法（例如 `whereOr` / `whereExp` 等）都不再支持传入 `Query` 对象。

### 取消 `resultset_type` 配置参数

数据集查询结果不再受 `resultset_type` 配置参数影响，默认情况下，Db查询统一返回数组，模型查询统一返回模型对象和模型数据集对象。如果Db查询的时候也需要返回数据集的话，可以显式调用 `fetchCollection` 方法。

### 取消 `Query` 类的 `extend` 方法

取消了 `Query` 类的 `extend` 方法，如果需要扩展查询方法，建议自定义 `Query` 类并继承系统的 `think\db\Query` 类即可，然后在模型中定义 `query` 属性或者配置数据库连接的 `query` 参数为你的自定义类。

### `Expression` 对象调整

原来的 `Expression` 对象已经更改为更适合的 `Raw` 对象，但不影响 `Db::raw()` 方法的调用。

### 取消查询 `eq/neq/gt/lt/egt/elt` 表达式

由于存在两种用法，并且不够直观，全部统一为更直观的用法。

下面的用法不再支持

```
Db::name('user')->where('id', 'egt', 1)
    ->where('status', 'neq' ,1)
    ->select();
```

统一使用

```
Db::name('user')->where('id', '>=', 1)
    ->where('status', '<>' ,1)
```

```
    ->select();
```

### 取消分表功能

出于分表的性能问题和复杂性，不再提供分表方法，建议使用数据库的分区功能替代。新版可以使用 `partition` 方法指定当前查询的分区。

### 数据库的查询统计合并

数据库的查询次数合并到 `queryTimes` ，不再区分读写操作，你可以使用下面的方法获取当前请求的数据库查询次数（包括读写）

```
Db::getQueryTimes();
```

### 模型后缀

如果之前开启了类库后缀功能的话，你必须在模型类里面明确指定 `name` 属性。

### 取消了模型的 `get` / `all` 方法

无论使用 `Db` 类还是模型类查询，全部统一使用 `find` / `select` 方法，取消了之前模型类额外提供的 `get` / `all` 方法。同时取消的方法还包括 `getOrFail` / `allOrFail` 。

### 取消全局查询范围 `base` 方法

取消模型类的全局查询范围 `base` 方法，改由使用 `globalScope` 属性定义（数组）需要全局查询的查询范围方法。

### 模型事件调整

模型事件不再需要使用 `event` 方法注册事件，统一在模型类中定义事件方法，例如

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

use think\Model;

class User extends Model
{
    public function onAfterRead($user)
    {
        $user->extra = 'extra';
    }

    public function onBeforeWrite($user)
    {
```

```
        $user->extra = 'extra';
    }
}
```

并且模型增加 `after_read` 事件，在查询后创建模型对象实例的时候触发。

### 取消模型自动完成

模型的自动完成功能已经取消，请使用模型事件代替。

### 模型 `save` 方法调整

模型类的 `save` 方法不再支持 `where` 参数。

### 关联统计调整

如果你的关联统计使用了闭包方式返回关联统计字段，需要调整为如下方式：

```
User::withCount(['cards' => function($query,&$name) {
    $query->where('status', 1);
    $name = 'card_count';
}])->select();
```

### 模型和数据集的输出调整

取消 `hidden` / `visible` / `append` 方法的第二个参数，当你调用这几个方法的时候，无论模型是否设置了相关属性，都会直接覆盖之前设置的值。

### 查询缓存调整

如果希望在更新和删除之后自动清除之前的查询缓存，必须在 `cache` 方法中传入key值而不是 `true` 。

### 删除关联类 `selfRelation` 方法

如果你在定义关联的时候使用了 `selfRelation` 方法，请直接删除该方法，目前已经不再需要，会自动识别是否为自关联。

### 删除关联类的 `setEagerlyType` 方法

一对一关联无需在定义关联的时候指定为 `JOIN` 查询，在查询的时候直接使用 `withJoin` 方法即可使用 `JOIN` 方式进行关联查询。

### 多对多关联

多对多关联的 `pivotDataName` 方法更名为更简单的 `name` 方法。

## 第六步：行为调整

行为和 `Hook` 已经用新版的事件机制替代，需要把你的行为改成事件响应或者中间件（部分请求拦截的行为可以直接改为中间件）。

原来的系统内置钩子的行为类

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

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

可以改成事件监听类

```
namespace app\index\listener;

class Hello
{
    public function handle($event)
    {
        // 事件监听处理
    }
}
```

然后在应用目录的 `event.php` 文件中配置事件监听。

```
return [
    'listen'    =>    [
        'AppInit'    =>    ['\app\index\listener\Hello'],
        // 更多事件监听
    ],
];
```

修改完成后，你可以删除应用目录下不再使用的 `tags.php` 文件。

内置事件和钩子的对应关系如下：

| 事件 | 对应 `5.1` 钩子 | 参数 |
| --- | --- | --- |
| `AppInit` | `app_init` | 无 |
| `AppEnd` | `app_end` | 当前响应对象实例 |
| | | |

| | | |
|---|---|---|
| `LogWrite` | `log_write` | 当前写入的日志信息 |
| `LogLevel` | `log_level` | 包含日志类型和日志信息的数组 |

> 原来的 `app_begin` 、 `response_send` 、 `response_end` 、 `action_begin` 、 `module_init` 和 `view_filter` 钩子已经废弃。

## 第七步：其它调整及注意事项

### 系统 `Facade` 类库别名取消

系统 `Facade` 类库的别名已经取消，因此不能再使用

```
use Route;
Route::rule('hello/:name', 'index/hello');
```

必须使用

```
use think\facade\Route;
Route::rule('hello/:name', 'index/hello');
```

### `Session` 调整

`Session` 新版默认不开启，必须为在全局中间件定义文件中添加

```
'think\middleware\SessionInit'
```

原来的 `Session::get()` 可以获取全部的Session数据必须改成 `Session::all()`

### 严格类型检查

由于新版框架核心类库全面启用强类型参数，并且使用严格模式，所以在调用系统方法的时候一定要注意参数的类型，或者注意看抛出的异常信息进行修正。

> 最后，希望你的 `6.0` 升级之旅顺利^_^ ，希望大家在升级和使用 `6.0` 的过程中，多反馈和建议，帮助我们尽快完善和发布正式版本。

# 更新日志

# 版本更新日志

- `V6.0.2` （2020年1月13日）
- `V6.0.1` （2019年12月24日）
- `V6.0.0` （2019年10月24日）
- RC5（2019年10月13日）
- RC4（2019年8月16日）
- RC3（2019年6月3日）
- RC2（2019年4月22日）
- RC1（2019年2月14日）

## `V6.0.2` （2020年1月13日）

本次更新包含一个可能的 `Session` 安全隐患修正，建议更新。

主要更新：

- 改进设置方法后缀后的操作名获取问题
- 修正 `optimize:schema` 指令
- 修正 `Request` 类 `inputData` 处理
- 改进中间件方法支持传多个参数
- 修正 `sessionid` 检查的一处隐患
- 完善对15位身份证号码的校验
- 增加远程多对多关联支持
- 增加 `MongoDb` 的事务支持（ `mongodb` 版本V4.0+）
- 改进 `insertAll` 的 `replace` 支持

## `V6.0.1` （2019年12月24日）

主要更新：

- 完善 `Request::withInput`
- 修正Content-Type获取途径
- 改进 SocketLog 驱动
- 修正 ClientArg 获取途径
- 修复FileResponse的cookie空对象异常

- 支持渲染完整的异常链信息
- 异常页面支持折叠调用堆栈信息
- 改进异常响应时内容的一致性
- 改进Error控制器对数字访问的支持
- 修正redirect助手函数
- memcached delete 支持 timeout
- 修正redis驱动
- 改进memcache驱动
- 改进容器类 `invokeMethod` 方法
- 使用新的数组语法替代list
- 缓存默认的序列化方法改为serialize/unserialize
- Add Cookie SameSite(PHP>7.3)
- 扩展 `Socket` 驱动
- 修正异常页面的模板文件
- 事件监听去重
- 取消视图日志
- 修正验证jpg结尾的图片报错问题
- 改进Url生成
- 改进Url生成伪静态后缀设置false的情况
- 改进File类型session读取 `gzcompress` 问题
- 修复使用路由验证后路由变量丢失的问题
- 修正多应用下路由目录路径
- 修复 `expand_level` 选项异常
- 改进Cache类 `remember` 方法对依赖注入的支持
- 防止因日志配置异常时陷入死循环
- 模型支持动态设置数据库连接
- 修正 `column` 方法的查询缓存问题
- 改进Query类的 `getAutoInc` 方法
- 改进模型更新条件获取
- 修正使用模型对象更改数据时忽略自定义的 `suffix` 和 `connection` 参数
- 修正XA事务
- 规范 `column` 方法的查询

## `V6.0.0` （2019年10月24日）

### 主要更新日志

- mobile 验证简化
- 控制器中间件支持传参
- 增强中间件CheckRequestCache的实用性
- 改进容器invokeClass方法异常处理
- 控制器中间件过滤条件支持字符串
- 改进Route类getDomainBind方法
- 防止路由标识生成URL时生成空字符串
- 让中间件CheckRequestCache只访问一次临界区
- 完善命中判断
- 修正Request::port返回类型
- 增强高并发下session_id生成的唯一性
- 取消路由缓存功能及相关指令
- 修正 Request::remotePort 返回类型
- 完善跨域中间件的响应头部
- 改进容器异常处理
- 改进生成文件模板
- 移除build指令纳入扩展
- 改进Response增加cookie方法用于设置Cookie
- 修正生成URL不传参数时，方法名为空
- 修复中间件CheckRequestCache使用过期数据
- 修复memcache驱动inc函数重复添加前缀问题
- 删除Resonse类setCookie方法
- 使用session_create_id生成id，保证唯一性
- 修复中间件FormTokenCheck无法启用问题
- 修正halt助手函数

## RC5（2019年10月13日）

`RC5` 版本主要改进包括多应用模式独立，以及中间件机制调整。

### 主要新特性

- 多应用模式独立为扩展
- 中间件分组执行
- 增加路由配置文件的全局中间件定义
- 中间件支持优先级定义
- 中间件支持请求结束回调

- 中间件执行去重
- 模板根目录免配置自动识别
- 改进智能事件订阅
- 页面Trace独立为扩展

## 更新日志

- 修正request类isJson方法
- 提升swoole下多应用的性能
- 修正Cookie保存
- 改进字段缓存指令
- 修正路由正则生成
- 改进模板变量的全局赋值
- 改进filesystem
- 改进View 增加单元测试
- 修正php模板驱动
- 取消view_base配置，增加view_dir_name配置
- 完善Session及其单元测试
- 改进session助手函数支持获取所有数据
- 完善中间件及其单元测试
- 改进事件智能订阅
- 验证错误信息支持数组
- 改进多语言自动侦测
- 改进自动多语言
- 改进事件观察者支持事件前缀
- 去除Cache返回类型限制
- 多应用配置读取优先级调整
- 多应用路由改进
- 改进批量验证的错误信息返回
- 页面Trace中间件移除 改为扩展
- 改进session初始化中间件
- 修正默认URL访问
- 修正url生成对域名绑定的支持
- 改进中间件机制 全局、路由和控制器中间件分开

## 调整

- 获取当前应用名改为 `app('http')->getName()`

## RC4（2019年8月16日）

`RC4` 版本主要改进包括 `ORM` 库独立，日志系统增加多通道支持，缓存、日志、数据库的配置文件统一调整为多通道模式，并作了大量的改进和修正。

### 主要新特性

- 数据库和模型改为独立的 `think-orm` 库
- 模型关联功能增强
- 日志支持多通道、并统一命令行和WEB日志格式
- 增加 `Filesystem` 组件
- 增加容器对象实例化回调机制
- 路由注解独立为 `think-annotation` 库
- 多应用模式下路由定义支持纳入应用目录

### 功能改进

- 改进路由类的配置读取
- 改进 `parseLike` 查询
- 改进 `ViewResponse` 类增加内容渲染输出支持
- 增加 `display` 助手函数
- 验证类 `maker` 方法支持注入扩展验证规则
- 改进事件触发的对象传入
- 改进 `Console/Table` 类
- 改进关联定义对查询构造器的支持
- 关联类增加 `withField` 和 `withLimit` 方法，并取消 `Query` 类 `withField` 方法
- 增加延迟关联查询对 `withLimit` 的支持
- 改进模型 `toArray` 方法
- 改进Url生成的域名参数
- 改进 `make:command` 指令生成
- 缓存有效期支持 `DateInterval`
- 改进Query类find方法
- 改进 `json` 查询
- 改进查询缓存
- 增加 `filesystem` 组件
- 改进跨域请求

- 改进路由检测缓存配置
- 改进注解路由的文件写入
- 路由注册支持注册 `options` 请求类型
- 改进重定向路由检测
- 改进模型的 `hasWhere` 方法对闭包查询条件的支持完善
- 增加 `time_query_rule` 数据库配置参数 用于自定义时间查询规则
- 改进时间字段类型的自动识别
- 改进 `redis` 驱动
- 容器对象增加 `invoke` 回调机制
- 改进多对多关联
- 资源路由增加 `withModel` 和 `withValidate` 对各个路由设置不同的模型绑定和验证
- 改进 `getLastInsID` 方法
- 日志类调整 支持多通道写入
- 改进模型输出
- 支持单独关闭某个通道的日志写入
- 取消部署模式下 不写入调试日志的功能 用调试级别设置单独的日志渠道来替代
- 改进日志记录格式
- 支持日志输出格式化
- 改进资源路由
- 改进命令行日志实时写入
- 日志通道支持单独设置level配置参数
- Log类调整优化
- 增加 `paginateX` 查询用于大数据分页查询
- 数据集增加 `first` 和 `last` 方法
- 改进 `response/View` 类的 `assign` 方法
- 改进请求参数获取问题
- 改进 `pathinfo` 方法
- 增加一些路径助手函数
- 改进多级控制器访问
- Log类支持 `__call` 方法
- 多应用模式的路由定义支持放入单独的应用目录
- 模型增加依赖注入支持
- 改进模型事件
- 改进页面 `Trace` 机制
- 改进 `validate` 助手函数,支持设置验证失败后是否抛出异常

- 改进容器 `bind` 方法
- 改进Redis Session驱动
- 改进日志记录的空行问题
- 改进 `RedirectResponse`
- 改进缓存驱动
- 改进 `think optimize:schema` 指令
- Url类增加https方法
- 改进 `isPjax` 判断
- 改进Db类配置获取

## 问题修正

- 修正模型属性获取
- 修正Request类的过滤功能
- 修正 `subDomain` 方法
- 修正 `input` 助手函数
- 修正模型 `refresh` 方法
- 修正关联统计不使用子查询的方式
- 修正Request类 `root` 方法
- 修正缓存有效期处理
- 修正 `MorphTo` 关联
- 修正缓存标签
- 修正mysql驱动 `insert` 方法
- 修正Db类 `connect` 方法
- 修正 `allowCrossDomain` 方法
- 修正Query类 `chunk` 方法
- 修正分组跨域
- 修正关联预载入查询
- 修正时间字段写入
- 修正验证类 `checkSize`
- 修正多对多关联闭包
- 修正RuleName类 `setRule` 方法
- 修正 `whereTime` 查询
- 修正 `request` 助手函数
- 修正日志关闭配置
- 修复对多字节字符的兼容性

### 用法调整

- 模型切换后缀方法 `switch` 更改为 `suffix`
- 取消Query类的 `fetchArray` 方法
- `select` 查询方法默认返回数据集对象
- 取消 `optimize:config` 指令
- 调整数据库和缓存配置文件格式 默认采用多类型支持 方便切换
- Cache类的 `init` 和 `connect` 方法取消 并入 `store` 方法
- `cache` 助手函数调整
- `cache` 助手函数取消初始化用法
- `cache` 助手函数留空返回Cache对象
- 调整调试模式检测位置
- 默认时间字段类型改为 `timestamp`
- 取消Model类 `getConnection` 和 `setConnection` 方法
- 注解路由移出核心，注解相关功能使用 `topthink/think-annotation`
- 改进路由 取消 `url_convert` 配置参数
- 取消 `route:build` 指令
- 调整默认模板目录为根目录 `view`
- 取消默认的请求日志记录 在项目里面自己添加
- 统一 `find` 查询必须使用查询条件
- 扩展的service配置文件默认放到vendor目录下
- 废除 `LogLevel` 事件
- 取消App类的序列化方法
- 控制台的 `user` 配置改为通过静态方法设置当前执行用户

## RC3（2019年6月3日）

RC3版本主要改进和优化了系统内置中间件，改进了一些用法和体验，并且把不常用的驱动移出核心改为扩展方式提供。

### 主要新特性

- 增加 `whereWeek` 日期查询
- 自增ID获取支持类型自动转换
- 当前请求记录匹配路由规则
- 增加 `requireWithout` 验证规则
- 优化路由ext和name方法以及URL生成
- 增加项目自定义类

- URL生成使用对象方式操作
- 修改器改进
- `dump` / `halt` 助手函数调整支持输出多个变量
- 关联自动更新
- 模型数据集增加 `delete` 和 `update` 方法
- 模型支持表后缀以及动态切换
- 注解路由支持给某个路由指定所属分组
- 多语言支持分组定义
- 支持自定义加载语言文件
- 多语言定义支持YML格式
- 缓存标签改进
- 缓存类增加 `push` 方法
- Cookie保存时间支持DateTimeInterface
- 增加表单令牌中间件
- 控制器支持__call方法
- 增加 `deny_app_list` 配置参数
- 控制器中间件only和except定义不区分大小写
- `app_map` 支持指定泛应用映射

## 问题修正

- 修正参数绑定的浮点型精度问题
- 修正软删除
- 修正模型的数据库连接
- 修正 `RedirectResponse`
- 修正Session类 `flush` 方法
- 修正JSON字段参数绑定
- 修正 `make:controller` 指令生成
- 修正Cache类的 `get` 方法默认值
- 修正域名绑定
- 修正关联模型的动态获取器
- 修正模型 `dateFormat` 属性方法
- 修正url生成对多入口的支持
- 修正ini配置文件格式的布尔值转换问题
- 修正路由延迟解析全局配置无效的问题
- 修正路由缓存问题

- 修正关联 `update` 操作
- 修正 `Relation::$selfRelation` 默认为 `null` ，导致 `Relation::isSelfRelation()` 方法报错
- 修正 `redis` 缓存驱动
- 修正事件智能订阅 `observe` 方法
- 修正模型字段定义对日期查询无效的问题
- 修正Console类 `getNamespaces` 方法
- 修正 `where` 查询方法传入Query对象的时候缺少 `bind` 数据的问题
- 修正request类 `method` 方法
- 修正 `route:list` 指令
- 修正 `Collection` 类 `load` 方法
- 修正 `redis` 驱动的端口类型
- 修正 `session` 数据序列化使用JSON处理的问题
- 修正分组路由合并解析
- 修正模型的 `hidden` 方法隐藏关联模型的问题
- 修正关联查询关联键为空的错误
- 修正返回204状态码的响应判断
- 修正 `Request` 类 `has` 方法对 `env` 和 `session` 的支持
- 修正 `provider.php` 文件无效问题
- 修正关联查询的部分问题
- 修正validate助手函数支持指定验证器类
- 修正验证类 `getValidateType` 方法
- 修正入口单独开启调试模式
- 修正加载Composer应用

## 用法调整

- 页面trace中间件仅在调试模式有效
- `Socket` 日志驱动移出核心
- `PostgreSQL` 、 `Sqlite` 和 `SqlServer` 驱动移出核心，改为扩展
- 取消内置 `think\Controller` 基类
- `Yaconf` 支持移出核心 纳入扩展 `think-yaconf`
- 字段排除改为 `withoutField` 方法
- 取消 `useGlobalScope` 方法增加 `withoutGlobalScope` 方法
- 更改默认生成的中间件位置

- 加载默认语言包无需开启多语言中间件
- `Cookie` 类恢复 `get` 和 `has` 方法支持
- `token` 助手函数调整
- 全局请求缓存参数调整
- 统一中间件调用传参，不支持 `:` 分割传参
- 缓存数据统一进行序列化后存储
- Cache类rm方法更改为delete方法
- validate助手函数返回Validate对象实例，参数改变

### 废弃用法

- 取消多语言的 `auto_detect` 配置
- 取消 `session` 类的 `auto_start` 配置参数和 `boot` 方法
- 废弃 `Where` 数组对象查询
- 取消模型事件观察者
- 取消 `JumpResponse` 及 `success` / `error` / `result` 等方法和助手函数
- 取消表达式查询解析扩展及 `think\db\Expression` 类
- 废弃模型自动完成功能，使用模型事件替代
- 取消 `cookie` 的 `prefix` 参数
- 取消一系列不推荐使用的助手函数
- 取消 `optimize:facade` `optimize:model` 指令 改为扩展方式
- 取消命令行执行URL
- 删除 `Config` 类 `__get` 和 `__isset` 方法

## RC2（2019年4月22日）

相比较 `RC1` 版本更新调整较大，主要更新如下：

- 底层架构针对协程做优化调整
- 增加WEB应用管理类 `Http`
- 增加应用初始化服务和服务注册机制
- 查询事件和模型事件使用事件系统接管
- `Session` 类重构，不再使用PHP内置会话
- `Cookie` 类仅支持设置和写入，不再支持读取
- 取消 `Config` 类的动态设置功能
- 部分核心功能中间件化（页面Trace、多语言、请求缓存和Session初始化）
- 取消惯例配置文件
- 增加驱动接口规范驱动开发

- 改进Db类和查询类
- 精简一些不必要的类库
- 改进路由注册
- 由于异常的需要 `View` 类改为内置，但默认仅支持原生PHP模板
- 大量细节改进和修正

## RC1（2019年2月14日）

- 改进Url类 `build` 方法
- 修复获取当前页码数据类型
- 修正 `parseKey` 方法传入数值的情况
- 改进 `optimize:config` 指令对 `declare` 申明的支持
- 取消URL参数模式配置
- 增加 `optimize:facade` 指令用于生成 `facade` 类的方法注释
- 取消 `Query` 类的 `extend` 方法
- 原来的 `Expression` 类更改为 `Raw` 类
- 增加新的 `Expression` 类用于表达式查询扩展
- `Collection` 类增加 `whereLike` / `whereNotLike` / `whereIn` / `whereNotIn` / `whereBetween` / `whereNotBetween` 等快捷方法
- Query类的 `raw` 方法移动到Db类
- 取消 `Request` 类的 `hook` 方法
- 修正 `Route::view` 方法
- 优化模型获取器方法
- 多对多关联的 `pivotDataName` 方法更名为 `name` 方法
- Query类增加 `partition` / `duplicate` / `extra` 方法
- 改进mysql驱动支持分区和 `duplicate` 以及额外参数
- mysql驱动的 `insert` 方法改用更清晰的 `insert set` 语法
- 修正 `Event` 类 `bind` 属性定义
- 修正验证类的 `append` 一处bug
- 改进 `Query` 类 `update` 方法支持读取模型的更新条件
- 取消模型类的 `getUpdateWhere` 方法统一使用 `getWhere` 方法
- 改进query类的 `fetchArray` 方法处理
- 改进路由类取消 `app` 属性
- 增加 `think\facade\RuleName` 类
- 增加 `whereFieldRaw` 查询方法
- 改进自动多应用名称获取

- App类增加 `withEvent` 方法支持关闭事件机制
- 改进 `Dispatch` 类对 `var_dump` 的支持
- 改进 `hasMany` 的 `withCount` 自关联
- 修正纯数字检测参数类型转换问题
- 修正 `raw` 助手函数
- `mysql` 支持 `find_in_set` 查询
- 改进 `url` 方法对自动多应用的支持