

# BroPHP 1.0 手册

## 目录

1 简介 .....	1
2 环境要求 .....	1
3 系统特点 .....	2
4 目录结构 .....	2
5 单一入口 .....	3
5.1 概述 .....	3
5.2 编写规则 .....	3
6 部署项目应用目录 .....	4
6.1 概述 .....	4
6.2 部署方法 .....	5
7 URL 访问 .....	9
8 默认开启 .....	10
9 配置文件 .....	10
10 内置函数 .....	11
11 控制器 .....	12
11.1 控制器的声明（模块） .....	12
11.2 操作的声明 .....	14
11.3 页面跳转 .....	15
11.4 重定向 .....	16
12 模型 .....	17
12.1 BroPHP 数据库操作接口的特性 .....	17
12.2 切换数据库驱动 .....	18
12.3 声明和实例化自己定 Model 类 .....	19
12.4 数据库的统一操作接口 .....	21
13 视图 VIEW .....	39
13.1 切换模板风格 .....	40
13.2 模板文件的声明规则 .....	40
13.3 display() 用新用法 .....	40
13.4 在模板中可以直接使用的几个常用变量 .....	41
13.5 在服务器中可以直接使用的几个常用变量 .....	41
14 自动验证 .....	42
15 缓存设置 .....	44
16 调试模式 .....	45
17 内置扩展类库 .....	46
17.1 分页类 Page .....	46
17.2 验证码类 Vcode .....	48
17.3 图像处理类 Image .....	49
17.4 文件上传类 FileUpload .....	50
18 自定义扩展函数库 .....	51
19 自定义扩展类库 .....	51



## 1 简介

**BroPHP(1.0)**是一个免费开源的轻量级 PHP 框架，允许你把基于 **BroPHP** 框架开发的应用去开源或是商业产品发布或销售。**BroPHP** 框架完全采用面向对象的设计思想，并且是基于 **MVC** 的三层设计模式。具有部署和应用极为简单、效率高、速度快，扩展性和可维护性都很好等特点，可以稳定的用于商业及门户的开发。（单服务器的 PHP 项目大约当前所有网站数量的 80%，都可以使用 **BroPHP** 框架开发）**BroPHP** 框架包括单入口文件、MVC 模式、目录组织结构、类自动加载、强大基础类、URL 处理、输入处理、错误处理、缓存机制、扩展类等功能。是专门为《细说 PHP》的读者及 LAMP 兄弟连全体学员提供的“学习型 PHP 框架”。当然任何 PHP 应用开发爱好者都可以从 **BroPHP** 框架的简单和快速的特性中受益。另外，**BroPHP** 框架的应用不仅使 WEB 开发变得更简单、更快捷，最主要的目的是让 PHP 学习者，通过使用本框架从而去了解 PHP 框架、再去研究框架，最后达到开发自己框架的目的。



可以到 <http://www.brophp.com> 和 <http://bbs.lampbrother.net>(LAMP 兄弟连)中下载 **BroPHP** 框架最新版本和最新的帮助文档。

LAMP 兄弟连：

高海峰

## 2 环境要求

**操作系统：** 支持 Linux/Windows 服务器, 可以跨平台应用  
**WEB 服务器：** 可运行于 Apache、IIS 和 nginx 中  
**PHP 环境：** PHP5.0 以上版本，需要安装 XML、mysqli、PDO、GD、Memcache 扩展模块

*PHP 新手推荐使用集成开发环境 AppServ 对 BroPHP 进行本地开发和测试*



## 3 系统特点

BroPHP 是“学习型”的超轻量级框架（文件很小，对 CPU 和内存消耗极低），虽然功能不算很多，但具备了一个框架构成最少应该有的全部功能（包括：MVC 模式、目录组织结构、类自动加载、基类、URL 处理、输入处理、错误处理、扩展类等）。本框架在已有的功能上，不管从组织结构上，还是从代码质量上，以及运行效率上都做到了单服务器最佳的效果。使用 BroPHP 框架适合开发 BBS、电子商城、SNS、CMS、Blog、企业门户等中小型系统。另外，本框架特别适合学习 PHP 使用，可以让你认识框架、分析框架内幕、从而达到编写自己框架的目的。并能通过框架的编写将你零散的各个 PHP 知识点组织在一起应用，BroPHP 框架也将《细说 PHP》中各章节知识点整合了起来。在分析框架源码时，PHP 的技术点可以参考《细说 PHP》各章节。BroPHP 框架部分特点如下：

1. 第一次访问时为用户自动创建了项目所需要的全部目录结构，用户无需在对组织项目的目录结构而烦恼。
2. 本框架采用模块和操作的方式来执行，简单易用，功能适中，更符合中国 WEB 程序员的开发习惯
3. 通过本框架编写的项目是完全采用 PHP 面向对象的思想，符合人类的思维模式，具有独立性、通用性、灵活性，有利于对项目的维护和调试。
4. 基于 MVC 的开发模式，将视图层和业务层的分离，达到快速的部署，具有很好的可维护性，以及高重用性和可适用性，特别有利于软件工程化管理。
5. 内建丰富的 SQL 查询机制，操作灵活，简单易用。
6. 采用了目前业界最著名的 PHP 模板引擎 Smarty，对于 Smarty 熟悉的程序员具有很好的模板开发优势。
7. 使用 Memcached 对 SQL 和 session 进行缓存，使用 Smarty 缓存技术进行页面静态化，提升效率，减少运行消耗。
8. 框架提供一些常用的扩展类，直接使用即可完成一些常见的功能。
9. 框架支持自定义扩展类库和扩展函数的使用
10. 采用人性化的调试模式，可以快速解决项目开发时遇到的错误和异常。
11. 框架源码简单明了结构清晰，方便在工作中根据当前项目的需求对框架进行改造。

## 4 目录结构

下例为 BroPHP 框架的系统目录，在项目开发时直接将 brophp 目录及子目录的所有文件复制到项目根目录中即可，并不需要对这个框架源文件有任何修改，但在 Linux 操作中要注意，将这个框架目录及子目录的权限，设置运行 PHP 的用户有读的权限即可。

-- brophp 目录	#BroPHP 框架目录
-- bases 目录	#BroPHP 框架基础类存放目录
-- classes 目录	#BroPHP 框架扩展类存放目录
-- commons 目录	#BroPHP 框架通过函数和资源存放目录
-- libs 目录	#Smarty 模板引擎源文件存放目录



## 5 单一入口

### 5.1 概述

在使用 PHP 过程化编程时，每个 PHP 文件都能独立访问并运行，就像一个体育场有多个入口一样，需要在每个入口都要检票和安全检查。而采用单一入口模式进行项目部署和访问，无论完成什么功能，一个项目只有一个统一（但不一定是唯一）的入口，就像一个体育场如果只能从一个入口入场（程序是抽象的，一个入口和多个入口效率是一样的）控制起来则更灵活，几乎没有什么缺点。使用主入口文件部署项目的优点如下：

#### ➤ 加载文件方便

在编写和阅读过程化程序代码时，经常会遇到文件之间互相包含，其中包括 PHP 使用 include 包括函数库和公共资源文件，也包括在 HTML 中使用 <link> 和 <script> 加载 CSS 和 JavaScript 文件。项目越大，文件越多越让人感觉头疼，就像一张大网一样将文件交织在了一起，不容易找到头绪。而使用单一入口则解决这个难题，在项目应用中用到的任何一个文件，只要相对于单一入口文件的位置查找即可。

#### ➤ 权限验证容易

如果每个 PHP 文件都可以独立访问，在做用户权限验证时就需要对每个文件进行判断。而采用单一入口则只需要在一个位置进行判断即可。

#### ➤ URL 重写简单

如果每个 PHP 文件和不同目录下的 PHP 文件都可以独立访问，则在 Web 服务器中对 URL 进行重新编写时需要很多条规则。而采用单一入口则在 URL 重写时只需要简单的几条规则即可。

### 5.2 编写规则

例如：index.php

```
<?php
define("BROPHP", "./brophp");      #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
define("APP", ".");                #定义项目的应用路径（'/'可加可不加）
require(BROPHP.'./brophp.php');    #加载 BroPHP 框架目录下的入口文件
```

#### ➤ 入口文件名

可以自己定义名称，例如 index.php、admin.php、blog.php 之类。



## ➤ 入口文件的编写规则

在入口文件中必需编写三行代码，每一条都不能省略（如上）。

第一行：是定义 BroPHP 框架源代码所在的路径，这个路径要相对于入口文件的位置，常量名（BROPHP）是固定的不能改变。例如，框架目录和入口文件的位置相同 `define("BROPHP", "../brphp")`，如果将入口文件写在项目的 `admin` 目录下则是 `define("BROPHP", "../../brphp")`。另外，这个路径值最后不要加结尾的“/”。

第二行：是定义项目的应用路径，即自己写的应用到那个目录下就指定那个目录，常量名（APP）是固定的不能改变。例如，`define("APP", ".")`，如果想将项目写在当前项目路径下的 `admin` 目录下则可以 `define("APP", "../admin/")`。另外，这个路径最后结尾的“/”可以不加。

第三行：是加载 BroPHP 框架目录下的入口文件，是固定的定写法。

## 6 部署项目应用目录

### 6.1 概述

下例提供的是项目应用目录，只是默认的方式。项目的应用目录结构并不需要开发人员手动创建，只需要定义好项目的入口文件之后（参与入口文件），系统会在第一次执行的时候自动生成项目必须的所有目录结构。注意：在 Linux 操作系统中，开发阶段需要让运行 PHP 用户有可写的权限，而当项目上线运行时，只需要 `runtime` 目录及子目录和 `public/uploads` 目录需要运行 PHP 用户有可写的权限，其他目录只要让运行 PHP 用户具有可读权限即可。

-- index.php 文件	#主入口文件（可以使用其他名称，也可以放在其他位置）
-- config.inc.php 文件	#项目的配置文件
-- controls 目录	#声明控制器类的目录
-- common.class.php 文件	#默认控制器的基类（用于写权限）
-- index.class.php 文件	#默认控制器（提供参考）
-- models 目录	#声明业务模型类的目录
-- views 目录	#声明视图的目录（Smarty 模板存放目录）
-- default 目录	#默认模板存入目录（可以为项目提供多套模板）
-- xxx 目录	#特定模块自己创建的目录（xxx 和模块同名）
-- xxx.tpl	#为特定的操作自己定义的模板文件（xxx 和动作同名）
-- public 目录	#同一应用中公用模板存放目录
-- success.tpl 文件	#同一应用页面跳转提示模板
-- resource 目录	#当前项目模板的资源目录
-- css 目录	#当前项目模板的样式目录
-- js 目录	#当前项目模板的 javascript 目录
-- images 目录	#当前项目模板的图片目录
-- classes 目录	#用户自定义的扩展类目录
-- commons 目录	#用户自定义的扩展函数目录
-- functions.inc.php 文件	#用户将自定义的扩展函数都必须写在这个文件中



-- public 目录	#项目的所有应用公用的资源目录
-- css 目录	#项目的所有应用公用的 css 目录
-- js 目录	#项目的所有应用公用的 javascript 目录
-- images 目录	#项目的所有应用公用的图片
-- uploads 目录	#项目的所有应用公用的文件上传目录
-- runtime 目录	#项目运行时自动生成文件存放目录（可以随时删除）
-- comps 目录	#Smarty 模板编译文件存放目录
-- cache 目录	#Smarty 页面静态缓存目录
-- data 目录	#项目使用的数据库结构缓存目录
-- controls 目录	#控制器缓存目录
-- models 目录	#业务模型缓存目录
-- _index.php 文件	#文件锁，如果目录结构存在则不再重新生成

上例只是以入口文件为 `index.php`，并且应用目录和框架目录在同一级时，默认生成的目录结构。具体的每个目录和文件的作用，在应用时参与后面部分的详细介绍。

## 6.2 部署方法

在部署网站时，项目的目录结构往往由不同项目的应用而决定。使用 BroPHP 框架时，项目的应用目录(`controls`、`models`、`views`)和入口文件的位置，可以由不同项目的应用自己决定存放位置，而其他公用资源目录和配置文件（`classes`、`commons`、`public`、`runtime`、`config.inc.php`）必须同框架目录 `brophp` 在同一级。这里推荐几种部署应用目录结构的方法：

### ➤ 如果项目只有一个应用(推荐两种方式)

第一种：入口文件和应用目录与框架在同级目录。这种方式比较简单，只需要在入口文件的第二行将应用目录常量 `APP` 的值改成“.”或“./”，然后直接访问入口文件即可生成所有目录结构。如下所示：

入口文件名命名：`index.php`（可以改为其它名称）

```
<?php
define("BROPHP", "./brophp");           #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
define("APP", ".");                     #定义项目的应用路径（'/'可加可不加）
require(BROPHP.'./brophp.php');
```

### 自动生成的应用目录结构（都是同级的）

-- brophp 目录	#BroPHP 框架目录
-- index.php 文件	#主入口文件（可以使用其他名称，也可以放在其他位置）
-- config.inc.php 文件	#项目的配置文件
-- controls 目录	#声明控制器类的目录
-- models 目录	#声明业务模型类的目录
-- views 目录	#声明视图的目录（Smarty 模板存放目录）
-- classes 目录	#用户自定义的扩展类目录
-- commons 目录	#用户自定义的扩展函数目录
-- public 目录	#项目的所有应用公用的资源目录



```
|-- runtime 目录          #项目运行时自动生成文件存放目录（可以随时删除）
```

第二种：项目的应用目录放到自己定义的目录下，例如声明一个应用目录名为“home”，入口和其他资源与框架同级。只需要在入口文件的第二行将应用目录常量 APP 的值改成“./home”或“./home/”，然后直接访问入口文件即可生成所有目录结构。如下所示：

入口文件名命名：index.php（可以改为其它名称）

```
<?php
define("BROPHP", "./brophp");      #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
define("APP", "./home/");          #定义项目的应用路径（'/'可加可不加）
require(BROPHP.'./brophp.php');     #加载 BroPHP 框架目录下的入口文件
```

自动生成的应用目录结构（controls、models 和 views 在 home 目录下）

```
|-- brophp 目录          #BroPHP 框架目录
|-- index.php 文件       #主入口文件（可以使用其他名称，也可以放在其他位置）
|-- config.inc.php 文件  #项目的配置文件
|-- home                #自定义的项目应用目录
    |-- controls 目录    #声明控制器类的目录
    |-- models 目录      #声明业务模型类的目录
    |-- views 目录       #声明视图的目录（Smarty 模板文件存放目录）
|-- classes 目录        #用户自定义的扩展类目录
|-- commons 目录        #用户自定义的扩展函数目录
|-- public 目录         #项目的所有应用公用的资源目录
|-- runtime 目录        #项目运行时自动生成文件存放目录（可以随时删除）
```

### ➤ 如果项目分为前台和后台操作(推荐三种方式)

第一种：前台和后台的入口文件都和框架目录 brophp 在同一级目录中，后台的应用目录定义在 admin（可以改为其他名称）下。然后分别访问两个入口文件即可生成所有目录结构。如下所示：

前台入口文件名命名：index.php（可以改为其它名称）

```
<?php
define("BROPHP", "./brophp");      #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
define("APP", "./");              #定义项目的应用路径（'/'可加可不加）
require(BROPHP.'./brophp.php');     #加载 BroPHP 框架目录下的入口文件
```





后台入口文件名命名：admin.php （可以改为其它名称）

```
<?php
    define("BROPHP", "../brophp");           #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
    define("APP", "../admin/");               #定义项目的应用路径（'/'可加可不加）
    require(BROPHP.'/brophp.php');           #加载 BroPHP 框架目录下的入口文件
```

自动生成的应用目录结构（后台的 controls、models 和 views 在 admin 目录下）

```
|-- brophp 目录           #BroPHP 框架目录
|-- index.php 文件        #前台主入口文件（可以使用其他名称）
|-- controls 目录        #声明控制器类的目录（前台控制器目录）
|-- models 目录          #声明业务模型类的目录（前台模型目录）
|-- views 目录           #声明视图的目录（前台视图目录）
|-- admin.php 文件        #后台主入口文件（可以使用其他名称）
|-- admin                #自定义的后台项目应用目录
    |-- controls 目录    #声明控制器类的目录（后台控制器目录）
    |-- models 目录      #声明业务模型类的目录（后台模型目录）
    |-- views 目录       #声明视图的目录（后台视图目录）
|-- config.inc.php 文件   #项目的配置文件
|-- classes 目录          #用户自定义的扩展类目录
|-- commons 目录          #用户自定义的扩展函数目录
|-- public 目录           #项目的所有应用公用的资源目录
|-- runtime 目录          #项目运行时自动生成文件存放目录（可以随时删除）
```

第二种：前台入口文件和前台应用与框架目录 brophp 在同一级目录中，后台的入口文件和应用目录定义在 admin（可以改为其他名称）下。然后分别访问两个入口文件即可生成所有目录结构。如下所示：

前台入口文件名命名：index.php （可以改为其它名称）

```
<?php
    define("BROPHP", "../brophp");           #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
    define("APP", "./");                     #定义项目的应用路径（'/'可加可不加）
    require(BROPHP.'/brophp.php');           #加载 BroPHP 框架目录下的入口文件
```

后台入口文件名命名：../admin/index.php （可以改为其它名称）

```
<?php
    define("BROPHP", "../brophp");           #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
    define("APP", "../");                   #定义项目的应用路径（'/'可加可不加）
    require(BROPHP.'/brophp.php');           #加载 BroPHP 框架目录下的入口文件
```

自动生成的应用目录结构（后台的入口文件和 controls、models 和 views 在 admin 目录下）

```
|-- brophp 目录           #BroPHP 框架目录
|-- index.php 文件        #前台主入口文件（可以使用其他名称）
```





```

|-- controls 目录      #声明控制器类的目录（前台控制器目录）
|-- models 目录       #声明业务模型类的目录（前台模型目录）
|-- views 目录        #声明视图的目录（前台视图目录）
|-- admin             #自定义的后台项目应用目录
    |-- index.php 文件 #后台主入口文件（可以使用其他名称）
    |-- controls 目录 #声明控制器类的目录（后台控制器目录）
    |-- models 目录  #声明业务模型类的目录（后台模型目录）
    |-- views 目录   #声明视图的目录（后台视图目录）
|-- config.inc.php 文件 #项目的配置文件
|-- classes 目录      #用户自定义的扩展类目录
|-- commons 目录      #用户自定义的扩展函数目录
|-- public 目录       #项目的所有应用公用的资源目录
|-- runtime 目录      #项目运行时自动生成文件存放目录（可以随时删除）

```

第三种：前台和后台入口文件与框架目录 `brophp` 在同一级目录中，前台和后台的应用目录分别定义在 `home`（可以改为其他名称）和 `admin`（可以改为其他名称）目录下。然后分别访问两个入口文件即可生成所有目录结构。如下所示：

前台入口文件名命名：`index.php`（可以改为其它名称）

```

<?php
    define("BROPHP", "./brophp");      #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
    define("APP", "./home/");          #定义项目的应用路径（'/'可加可不加）
    require(BROPHP.'./brophp.php');    #加载 BroPHP 框架目录下的入口文件

```

后台入口文件名命名：`admin.php`（可以改为其它名称）

```

<?php
    define("BROPHP", "./brophp");      #定义 BroPHP 框架所在路径(相对于入口文件，不要加'/')
    define("APP", "./admin/");          #定义项目的应用路径（'/'可加可不加）
    require(BROPHP.'./brophp.php');    #加载 BroPHP 框架目录下的入口文件

```

自动生成的应用目录结构（后台的入口文件和 `controls`、`models` 和 `views` 在 `admin` 目录下）

```

|-- brophp 目录      #BroPHP 框架目录
|-- index.php 文件   #前台主入口文件（可以使用其他名称）
|-- home            #自定义的前台项目应用目录
    |-- controls 目录 #声明控制器类的目录（前台控制器目录）
    |-- models 目录  #声明业务模型类的目录（前台模型目录）
    |-- views 目录   #声明视图的目录（前台视图目录）
|-- admin.php 文件   #后台主入口文件（可以使用其他名称）
|-- admin           #自定义的后台项目应用目录
    |-- controls 目录 #声明控制器类的目录（后台控制器目录）
    |-- models 目录  #声明业务模型类的目录（后台模型目录）
    |-- views 目录   #声明视图的目录（后台视图目录）
|-- config.inc.php 文件 #项目的配置文件

```



-- classes 目录	#用户自定义的扩展类目录
-- commons 目录	#用户自定义的扩展函数目录
-- public 目录	#项目的所有应用公用的资源目录
-- runtime 目录	#项目运行时自动生成文件存放目录（可以随时删除）

### ➤ 项目有多个应用

如果项目有多个应用。例如，除了有前台和后台还有博客和论坛，每个应用都需要有独立的入口文件和自己的应用目录。可以让所有入口文件在同一级（例如：与框架在同级目录，但名称不能相同，可以和应用目录名称相同），也可以将每个入口文件放在自己的应用目录中（入口名称就可以统一命名为：index.php）。部署方式同上。

## 7 URL 访问

BroPHP 框架的 URL 都是使用 PATHINFO 模式（index.php/index/index/），应用的访问方式都是采用单一入口的访问方式，所以访问一个应用中的具体模块以及模块中的某个操作，都需要在 URL 中通过入口文件后的参数来访问和执行。这样一来，所有访问都会变成由 URL 的参数来统一解析和调度。格式如下：

http://www.brophp.com/入口文件/模块名/操作名/参数 1/值 1      #URL 统一解析和调度的 PATHINFO 模式

例如 1：项目应用代码直接放到主机为 [www.brophp.com](http://www.brophp.com) 的 Web 服务器的文档根目录下，入口文件名为 index.php，访问用户模块(user)，再去执行添加（add）的操作，则 URL 的格式如下：

http://www.brophp.com/index.php/user/add      #通过 URL 统一解析和调度

如果还需要其他参数，例如在上例添加数据时，需要将数据加到类别 ID 为 5 的类别(cid=5)中,则可以在上例 URL 的操作名后继续加多个参数。则 URL 的格式如下：

http://www.brophp.com/index.php/user/add/cid/5      #也可以有更多的参数

如果访问某个应用的入口时没有给出需要访问的模块和操作，则默认访问模块为 index,默认访问的操作为 index。下面几个 URL 的访问结果是相同的。如下所示：

<http://www.brophp.com/>  
<http://www.brophp.com/index.php>  
<http://www.brophp.com/index.php/>  
<http://www.brophp.com/index.php/index>  
<http://www.brophp.com/index.php/index/>  
<http://www.brophp.com/index.php/index/index>  
<http://www.brophp.com/index.php/index/index/>

如果访问某个应用的入口时只给出访问的模块名，没有给出访问模块中的动作名，则默认访问这个模块中的 index 操作。像如访问用户模块（user），下面几个 URL 的访问结果也是相同的。如下所示：

<http://www.brophp.com/index.php/user>  
<http://www.brophp.com/index.php/user/>  
<http://www.brophp.com/index.php/user/index>  
<http://www.brophp.com/index.php/user/index/>



如果在 URL 访问中除了模块和操作，还需要其他参数，就必须给出模块名和动作名（包括默认模块 `Index` 和默认操作 `Index`）全格式，再加上多个参数。如下所示：

<http://www.brophp.com/index.php/index/index/cid/5/page/6> #追加两个参数 `cid=5` 和 `page=6`

`PATHINFO` 模式对以往的编程方式没有影响，`GET` 和 `POST` 方式传值依然有效，因为系统会对 `PATHINFO` 方式自动处理。例如，上面 URL 地址中的 `cid` 的值可以通过 `$_GET['cid']` 的方式正常获取到。

## 8 默认开启

在入口文件的第三行 “`require(BROPHP.'/brophp.php')`”，我们加载 BroPHP 框架目录下的入口文件 `brophp.php`。在这个框架入口文件中有一些为整个应用默认开启的功能，所以在项目应用时就不需要再去设置了，没有必要的话默认的设置都不需要进行修改。如表所示：

表 BroPHP 默认开启的功能和描述

应用中默认开启功能	描述
输出字符集（utf-8）	utf8 是网站和 MySQL 数据库的最佳选择，没有必要请不要改变。
设置时区（PRC）	将 PHP 环境中的默认时间区改为中国时区。
自动加载项目的配置文件（ <code>config.inc.php</code> ）	整个项目只有一个匹配文件 <code>config.inc.php</code> ，在项目中被自动包含，所以在用到配置文件中的选项时，都可以直接使用。
自动包括类库和函数库	在应用中用到的所有类和函数都是自动包括的，项目开发时只要按规范去编写，都不需要去特意包含。
自动开启 Session	自动开启会话控制，如果启用 Memcache 则将用户的会话信息写入到 Memcache 服务器，否则使用默认的写入方式

## 9 配置文件

Web 项目几乎都需要有配置文件，这样才能更灵活的对项目进行管理和维护。BroPHP 框架在第一次访问时，为整个项目自动创建了一个配置文件 `config.inc.php`（仅一个），存放在与框架目录同级的目录中，并且默认被包含在程序中，所以在项目开发时配置文件中的选项都可以直接应用。另外，除了配置文件中默认选项可以直接使用以外，还可以自定义添加一些选项在这个文件中，自定义的选项可以是常量，也可以变量和数组等，如果添加的是变量或数组，则在所有编写的函数和类中需要使用 `global` 包含这些全局变量。系统自动创建的配置文件默认选项介绍如表：

表 BroPHP 框架配置文件的选项和描述

配置选项	描述
<code>define("DEBUG", 1);</code>	设置是否开启调试模式（1 开启，0 关闭），建议在开发时使用 1 值开启调试模式，上线运行则使用 0 值将其关闭。默认值为 1 开启。
<code>define("DRIVER","pdo");</code>	设置数据库的驱动选项，本系统支持 <code>pdo</code> （默认）和 <code>mysql</code> 两种驱动方式。在开启 <code>mysql</code> 时需要 PHP 环境安装 <code>mysql</code> 扩展模块，在使用 <code>PDO</code> 选项时，除了需要 PHP 环境中安装 <code>PDO</code> 的扩展模块外还需



	要安装相应数据库的驱动。
// define("DSN","mysql:host=localhost;dbname=xsphp");	当上面的 DRIVER 选项设置为 pdo 时，则可开启这个 PDO 的数据源设置，如果设置了此选项则可以不用再去设置以下的 HOST、USER、PASS 和 DBNAME 选项。
define("HOST", "localhost");	数据库系统的主机设置选项,默认为 localhost
define("USER", "root");	数据库系统用户名,默认为 root
define("PASS", "123456");	数据库系统用户密码，默认为空
define("DBNAME","brophp");	应用的数据库名称，默认为 brophp
define("TABPREFIX", "bro_");	设置数据表名的前缀，防止在相同的数据库中保存两个以上 BroPHP 框架开发项目的数据表。另外，这个选项同时也作为 Memcache 的键前缀，作用同表名前缀一样，防止同一个 Memcache 服务器有两个以上的 BroPHP 应用。
define("CSTART", 0);	这个选项用来设置 Smarty 的缓存，项目开发阶段使用 0 关闭缓存，在项目上线运行时设置 1 将其开启。默认为 0。
define("CTIME", 60*60*24*7);	这个选项设置 Smarty 模板的缓存时间，同时也是 Session 在 Memcache 中的生存时间。默认值为一周。
define("TPLPREFIX", "tpl");	Smarty 模板文件的后缀名，视图中所有 Smarty 模板的后缀名都要和这个相同，默认后缀名为 tpl。
define("TPLSTYLE", "default");	这个选项设置项目使用的模板风格。可以为一个项目开发多套模板风格，使用这个选项进行切换。默认使用的模板风格为 default
// \$memServers = array("localhost", 11211); // \$memServers = array( array("www.lampbrother.net", '11211'), array("www.brophp.com", '11211'), ... );	这个选项用来设置 Memcache 服务器的主机和端口，如果是一个一维数组则连接一个 Memcache 服务器，也可以是一个二维数组同时连接多个 Memcache 服务器。另外，如果注释没有打开则没有开启 Memcache 服务器，建议安装 Memcache 服务器并将其开启。

## 10 内置函数

BroPHP 框架在内部的 commons 目录下的 functions.inc.php 文件中，提供了几个常用的快捷操作的全局函数，并在 BroPHP 的入口文件 brophp.php 中自动包含了该文件，所以在任何位置都可以直接调用这几个函数。包括 P()、D() 和 toSize() 三个内置函数，详细的功能介绍和用法如下：

- **函数 P()**：按照特定格式打印输出一个或多个任意类型（数组、对象、字符串等）的变量或数据，打印的值供程序员作为开发程序时的参考使用。使用方式如下所示：

```
$arr=array(1,2,3,4,5,6);
```



```

P($arr);           #可以打印输出 PHP 数组
$object=new Object();
P($object);        #可以打印输出 PHP 对象
$string="this is a string";
P($string);        #可以打印输出 PHP 自己串
$other=其它类型;
P($other);         #可以打印输出 PHP 的任何类型
P($arr, $object, $string, $other); #可以同时打印输出多个 PHP 变量

```

- **函数 D()**：快速实例化 Model 类库，而且实例化 Model 类只能用这个函数。而且这个函数不仅可以实例化已声明的 Model 类，也可以实例化没有声明的 Model 类(只要参数对应的表名存在即可)。另外，不仅可以声明自己应用中的 Model 类，也可以声明其他应用中的 Model 类。该函数大量用于控制器中，使用方式如下所示：

```

$book = D("book");           #如果在本应用的 models 中声明了一个类 Book,则实例化 book 对象
$book = D("book");           #如果在本应用的 models 中没有声明 Book 类，但 book 表存在也行
$book = D("book", "admin");  #如果有第二个参数，可以实例 admin 应用下的 book 对象

```

- **函数 toSize()**：就是一个普通的功能函数，将字节大小根据范围转成对应的单位(KB、MB、GB 和 TB 等)，该函数只有一个参数就是字节数。

```

toSize(10240);           #结果返回 10KB。

```

## 11 控制器

BroPHP 框架是以模块和操作的方式来执行的，一个项目应用中会有多个模块，每个模块又需要单独去调度，所以控制器是一个模块的核心，**建议为每个模块单独声明一个控制器类。**

### 11.1 控制器的声明（模块）

控制器就是类似于我们平常所说的控制器，系统会自动寻找项目应用的 controls 目录下面的相关类，如果没有找到，则会定位到空模块，否则输出错误提示。例如，一个网上书店中有用户管理(user)、类别管理(cat)和图书管理(book)三个模块，则需要创建三个控制器 User 类、Cat 类和 Book 类和三个模块对应。访问一个模块中的控制器和控制器中的操作都需要通过入口文件完成，控制器会管理整个用户的执行过程，负责模块的调度和操作的执行。另外，任何一个 WEB 行为都可以认为是一个模块的某个操作，也需要通过入口文件来执行，BroPHP 系统中会根据当前的 URL 来分析要执行的模块和操作。在 URL 访问一节中介绍了 BroPHP 框架的 URL 访问格式。如下所示：

```

http://www.brophp.com/入口文件/模块名/操作名/参数 1/值 1      #URL 统一解析和调度的 PATHINFO 模式

```



例如：

<http://www.brophp.com/index.php/user/mod/id/5>

#URL 访问格式

上例是获取当前需要执行项目的入口文件（index.php）、模块（user）和操作（mod），如果有其他的 PATHINFO 参数（/id/5）会转成 get 请求（\$\_GET["id"]=5）形式。在这个例子中，用户访问的是 User 模块，就需要为这个模块定义一个控制器 User 类才能被调度。为模块的控制器在 BroPHP 中有专门的声明位置，是声明在当前项目应用目录下的 controls 目录中，类名必须和模块名相同，这个例子中使用 user 模块，就需要创建一个 User 类(每个单词的首字母要大写)保存在 user.class.php 文件中（文件名和类名相同，所有 BroPHP 中声明的类都要以.class.php 为后缀名）。如下所示：

```
/*
 * 定义在 controls 目录下，文件名为 user.class.php
 */
Class User {
    //声明控制器的操作
}
```

在自定义的控制器类 User 中，通常不需要去继承其他的类，如果写继承也只能继承 BroPHP 框架中的基础类 Action，不能有其他的继承方式。如下所示：

```
/*
 * 定义在 controls 目录下，文件名为 user.class.php
 */
Class User extends Action{
    //声明控制器的操作
}
```

在自定义的控制器类 User 中，如果不去继承系统中的 Action 类，则默认会继承控制器的通用类 Common。Common 类声明在 common.class.php 文件中，是部署项目应用时自动创建的一个文件，也保存在当前应用的 controls 目录下。Common 类的默认格式如下所示：

```
/*
 * 定义在 controls 目录下，文件名为 common.class.php
 */
Class Common extends Action{
    function init(){
        // 所有的操作都会先执行这个方法
        // 通过用于设置用户登录
    }

    //可以自定义一些方法，作为所有控制器的共用操作方法
}
```

用户自定义的控制器类（User）自动继承了 Common 类，而 Common 类又继承了 BroPHP 框架基础类中的 Action 类。所以在 User 类中就可以直接使用从 Action 类中继承过来的所有属性和方法。Common 类存在的目的有两个：





1. 在 `Common` 类中有一个默认的方法 `init()`，这个方法是所有操作执行前先执行的方法，所以可以在这个方法中完成用户的登录操作。
2. 在 `Common` 类中可以自定义一些方法，则可以被所有自动继承该类的控制器共用。

## 11.2 操作的声明

在上例的 URL 访问中，除了需要声明控制器 `User` 类之外，还需要在控制器 `User` 中定义用户的操作方法。每一个操作都对应当前模块控制器中的一个方法。例如，上例访问的模块是 `user`(对应 `User` 类)，而执行的动作 `mod`(对应 `User` 类中的 `mod` 方法)，如果后面还有其他 `PATHINFO` 参数，则将以 `get` 方式传给这个方法。如下所示：

```
/*
 * 在 controls 目录下，文件名为 user.class.php，默认继承 Common 及 Action
 */
Class User {
    //控制器中默认的方法，用于获取用户默认的操作
    function index(){
        // 这个方法的 URL 访问如下两种方式，可以不写操作名 (index)
        // http://www.brophp.com/user/
        // http://www.brophp.com/user/index
    }
    //控制器中声明的方法，用于添加用户的操作
    function add(){
        // 这个操作的访问如下，模块 (user)，操作 (add)
        // http://www.brophp.com/user/add
    }
    //控制器中声明的方法，用于修改用户的操作
    function mod(){
        // 这个操作的访问如下，模块 (user)，操作 (mod)
        // http://www.brophp.com/user/mod/id/5

        P($_GET["id"]);
        /*
            输出结果: Array("id"=>5);
        */
    }
    //控制器中声明的方法，用于删除用户的操作
    function del(){
        // 这个操作的访问如下，模块 (user)，操作 (del)
        // http://www.brophp.com/user/del
    }
    //控制器中私有的其他方法，不是一个操作
    Private function upload(){
        // 这个用于上传用户头像，不是操作则不能用 URL 访问
    }
}
```





## 11.3 页面跳转

自定义的控制器类直接或间接的继承 BroPHP 系统中的基类 Action，所以 Action 类内置了一些方法，并可以在每个控制器的方法中直接使用 \$this 访问。在应用开发中，经常会遇到一些带有提示信息的跳转页面，例如操作成功或者操作错误需要自动跳转到另外一个目标页面。页面跳转在 Action 类中提供了 success() 和 error() 两个方法，详细的使用方法如下所示：

### ➤ 成功操作跳转 success()

在进行添加或修改等的一些操作时，如果操作成功通常都会自动跳转到一个提示页面，然后再自动跳转到一个目标页面。Success() 方法是系统 Aaction 类内置的方法，用在自定义控制器的方法中。这个方法格式如下所示：

```
Success(提示信息, [跳转时间], [目标位置])
```

这个方法有三个参数，并且都是可选的。其中第一个参数用于在提示页面中输出成功消息，默认消息就是简单的“操作成功”的提示字样。第二个参数用于设置提示页面的停留时间，默认为 1 秒（时间很短，成功提示没有必要停留时间过长），可以通过传递一个整数重新设置这个时间（单位：秒）。第三个参数是自动跳转的目标位置（这个位置必须是 PATHINFO 的格式），如果只有一个字符串（index）指定目标方法，则表示自动跳转到同一个模块的这个方法中。如果使用“/”分开的字符串（模块/操作，例：user/index），则表示跳转到其他模块指定的操作中，也可以在这个参数中使用其他的参数将一些数据带到新的目标方法中，如果没提供第三个参数，则默认返回（window.history.back()）。常见的用法如下所示：

```
$this->success();           //默认方式
$this->success("添加成功");   //只有第一个参数
$this->success("添加成功", 3); //使用两个参数
$this->success("添加成功", 3, "user/index"); //使用三个参数
$this->success("添加成功", 3, "user/index/cid/5"); //可以加资源
```

成功的提示界面如下所示，可以根据自己的爱好对界面进行修改。在提示界面中，有停止跳转的操作，也可以手动“单击”跳转。





## ➤ 失败操作跳转 error()

在进行添加或修改等的一些操作时，如果操作失败更需要自动跳转到一个提示页面，查看出错原因，然后再自动跳转到一个目标页面。`error()`方法也是系统 `Aaction` 类内置的方法，也用在自定义控制器的方法中。这个方法的使用方式和 `success()`方法完全相同，只是提示界面和默认的提示消息及跳转时间不同而已。错误的提示界面如下所示：



## 11.4 重定向

如果某个操作（控制器中的方法）执行完成以后，需要转向到其它的操作，有时也需要将当前操作中的一些数据也带到另一个操作中，就可以使用从系统基类 `Action` 中继承过来的 `redirect()`方法实现，重定向后会改变当前的 URL 地址。例如，在 `User` 模块的控制器中，执行 `del` 操作成功删除用户后重定向到当前模块的 `index` 操作中。如下所示：

```
/*
 * 在 controls 目录下，文件名为 user.class.php，默认继承 Common 及 Action
 */
Class User {
    //控制器中默认的方法，用于获取用户默认的操作
    function index(){
        // 默认的操作方法
    }
    //控制器中声明的方法，用于删除用户的操作
    function del(){
        // 创建用户对象
        $user = D("user");
        //使用用户对象中的 delete 删除某一个用户
        if ($user->delete($_GET["id"])){
            //如果删除成功就重新定向到 本模块的默认的操作 index 中
            $this->redirect("index");
        }else{
            //如果删除失败就跳转到 提示界面，并返回
            $this->error("删除用户失败");
        }
    }
}
```



### ➤ redirect()方法的其他应用:

```
$this->redirect("模块/动作");    #如果有使用 "/" 分成模块和操作  
$this->redirect("book/add");    #重定向到 book 模块的 add 操作中（例）
```

如果在重定向到其他操作中时，还需要带一些参数过去，还可使用第二个参数以 PATHINFO 的形式将数据传过去。如下所示：

```
$this->redirect("模块/动作", "参数");    #使用第二个参数，传数据（PATHINFO 格式）  
$this->redirect("book/index","cid/5/page/3");    #传了 cid 和 page 两个参数（例）
```

上例在 redirect()中使用了第二个参数，在重定向到 book 模块的 index 方法中的同时，也将 cid=5 和 page=3 两个参数传到了 book 模块的 index 方法中。

## 12 模型

模型(Model): 就是业务流程/状态的处理以及业务规则的制定。业务流程的处理过程对它层来说是黑箱操作，模型接受视图请求的数据，并返回最终的处理结果。业务模型的设计可以说是 MVC 最主要的核心。在 BroPHP 中基础的模型类就是内置的 DB 类，该类完成了基本的数据表增、删、改、查、连贯操作和统计查询，以及一些高级特性都被封装模型类中。

### 12.1 BroPHP 数据库操作接口的特性

编写程序的业务逻辑最繁琐的地方就是对不同数据表反复写操作 SQL 语句（增、删、改、查），降低网站性能的最大开销就是在程序中执行 SQL 查询，攻击网站最常见的方式是使用 SQL 注入，所以在 BroPHP 框架的 Model 中解决了这些问题。

#### ➤ 重用性

BroPHP 内置了抽象数据库访问层，把不同的数据库操作封装起来，而使用了统一的操作接口。只需要使用公共的 Db 类进行操作，而无需针对不同的数据表写重复的代码和底层实现。

#### ➤ 高效性

BroPHP 框架的 Model 中，所有的 SQL 语句都是通过 prepare()和 execute()方法去准备和执行，效率要比使用 query()方法高得多。另外，最主要的是在 BroPHP 中所有的查询结果都使用 Memcache 进行缓存，所以只要获取一次结果集，同样的查询下次不管再执行多少次都不需要再重新连接数据库了，而是直接从 Memcache 中获取数据，这样可以大大提高网站的性能。并且如果有执行对表有影响的 SQL 语句就会清除该表的缓存，所以还可以达到动态更新的效果。

#### ➤ 安全性

每个 SQL 语句都是使用 mysqli 或 PDO 中的预处理方式，并通过“？”参数绑定的形式先将语句在服务器中准备好，再为这个“？”绑定的任何“值”都不会再重新编译 SQL 语句，所以 BroPHP 框架没有 SQL 注入的可能。

#### ➤ 简易性



BroPHP 框架为所有自定义 Model 类的实例化提供了统一的内置函数 `D()` 来实现，简单了 Model 类的对象创建过程。并且所有的 SQL 查询都可以采用连贯操作方式，以及使用系统中内置的方法就可以以最简单的方式完成对数据表的操作。

#### ➤ 扩展性

BroPHP 框架中 Model 类之前继承关系简单明了，很容易通过自己定义的 Model 类对系统中内置 DB 类的功能进行扩展，完成特定的功能。

#### ➤ 维护性

BroPHP 框架中所有和 SQL 语句相关的执行都会汇总到一个操作中，有统一的处理方式。这样就可以大大提高 Model 类的可维护性。

## 12.2 切换数据库驱动

BroPHP 框架支持 `mysqli` 和 `PDO` 两种连接方式的驱动，并且都是使用他们的“预处理”方式来处理 SQL 语句，这样不仅效率高而且能防止 SQL 注入。默认是使用 `PDO` 的连接方式（推荐使用 `PDO`，除了可以连接 MySQL 数据库还可以连接其他数据库）。不管理使用那种连接方式，在使用前要先安装 PHP 扩展库，`PDO` 还需要安装对应的数据库驱动。切换的方式也很容易，只要修改配置文件（和框架在相同目录的 `config.inc.php` 文件）中一个参数，DB 类就会自动调用相应的数据库适配器来处理。如下所示：

```
/*
 *项目的配置文件 Config.inc.php （和框架同目录下）
 */
define("DRIVER", "pdo"); // pdo(默认) 和可改成 mysqli
```

在 Model 中如果能正确的连接数据库，除了在配置文件中设置上例的选择数据库驱动方式，还需要配置数据库的连接用户和密码，以及数据库的库名。如下所示

```
/*
 *正确的配置数据库的连接
 */
define("HOST", "localhost"); // 数据库服务器的主机位置
define("USER", "root"); // 数据库服务器的登录用户
define("PASS", ""); // 数据库登录用户的密码
define("DBNAME", "brophp"); // 数据库的库名
define("TABPREFIX", "bro_"); // 数据表的名称前缀
```

如果选择 `PDO` 来连接数据库，还可以使用 `DSN`（数据源名）的方式来配置数据库的连接，这样的配置不仅可以连接 `PDO` 连接 MySQL 数据库，还可以连接其他数据库。也是通过在配置文件中修改配置，如下所示：

```
/*
 *使用 DSN 的方式配置 PDO 连接数据库
 */
define("DSN", "mysql:host=localhost;dbname=brophp"); //DSN 方式
```



如果使用 DSN 的方式配置数据库连接，则不用再去配置 HOST、USER、PASS 和 DBNAME 四个选项了，因为在 DSN 中都有设置了。

## 12.3 声明和实例化自己定 Model 类

所有对数据表的操作都需要使用 BroPHP 的 Model 完成，而不管管理是自己定义 Model 类(DB 类的子类)，还是直接使用系统内置的数据库操作类，都需要使用内置的 D()方法来实例化一个 Model 对象。

### ➤ 声明自定义的 Model 类

在配置文件中配置好与数据库连接有关的选项以后，就可以为数据表声明一个对应的 Model 类来处理它了，自定义的 Model 类名必须和数据表名相同（BroPHP 采用的是通过类名找对应的表处理）。例如，数据库中有三张表 bro\_books、bro\_users 和 bro\_articles（其中 bro\_为表名前缀，会自动处理），就需要在当前应用的 models 目录下创建 books.class.php、users.class.php 和 articles 三个文件（类名不用加表前缀名）。在每个文件中只声明一个对应的类，如果不去写继承会自动继承系统中的 DB 类，就可以直接使用从 DB 类中继承过来的内置方法操作数据表了。自定义的 Model 类 Users 如下所示：

```
/*
 * 在 models 目录下，文件名为 users.class.php，默认继承系统内置的 DB 类
 * Users 类是 DB 的子类，可以直接使用所有从 DB 类中继承过来的方法
 */
Class Users {
    //声明一个登录的方法
    function isLogin(){
        // 方法体
    }
    //声明一个退出系统的方法
    function isLogout(){
        // 方法体
    }
}
```

如果有两个 Model 类需要声明一个父子类，用于构建共用的属性和方法，当然也可以直接使用 extends 继承一个自定义的一个公用父类，但主动继承的父类也会自动继承系统内置的 DB 类。所以自定义的 Model 类还是间接的继承了 DB 类，这样除了可以直接使用自定义父类中的成员，还可以直接使用系统内置类 DB 中的成员。自定义的 Model 类 Books 继承自定义的 Demo 类，如下所示：

```
/*
 * 在 models 目录下，文件名为 Demo.class.php，将来会自动继承 DB 类
 */
Class Demo {
    //声明一个的方法
    function fun(){
        // 多个子类共用这个方法
    }
}
```



```

}
自己定义的 Model 类 Books 主动使用 extends 继承上例中的 Demo 类，如下所示：
/*
 * 在 models 目录下，文件名为 books.class.php
 * 声明一个类 Books 主动继承 Demo 类，Demo 类会自动继承系统内置的 DB 类
 */
Class Books extends Demo{
    //在这个类中可以使用 Demo 类和系统内置类 DB 中的所有继承过来的成员
}

```

在声明好一个 Model 类之后，就可以在当前项目应用的控制器中，使用系统内置的 D() 函数去实例化这个 Model 类的对象，再通过这个对象就可以直接对业务进行了。在使用 D() 函数时，需要提供一个参数，参数必须是自定义的 Model 类名（和表同名）。例如，声明好了一个 Users 模型类后，在 User 控制器使用过程如下所示：

```

/*
 * 在 controls 目录下，文件名为 user.class.php, 默认继承 Common 及 Action
 */
Class User {
    //控制器中默认的方法，用于获取用户默认的操作
    function index(){
        // 创建用户对象，参数 users 找模型中的 Users 类创建的
        $user = D("users");
        $data=$user->select();//调用父类中的方法查询表中所有记录
        //$user->isLogin(); 也可以调用 Users 类中声明的方法
    }
}

```

在使用 D() 函数实例化模型类时，系统会自动通过参数字符串找到对应的数据表。如果这个数据表是第一次操作，则系统会自动获取表结构并缓存一起来，以后的每次操作都是从缓存中直接获取表结构，不会每次都重新连接数据库反复获取表结构。

#### ➤ 直接使用内置 DB 类

如果只需使用系统内置 DB 类中的功能就可以完成对业务的处理，则没有必要单独声明一个空（没有成员）的 Model 类（只有需要对某个表有特定的操作，DB 类没有提供的功能，才去自定义 Model 完成一些特定的处理），也是使用 D() 函数实现。例如，没有声明对用户表(users) 操作的模型类时，使用 D() 直接传表名（不用加前缀）作为参数（用于获取表结构），就可以实例化一个 DB 类的对象，完成对 users 表的操作。如下所示：

```

/*
 * 在 controls 目录下，文件名为 user.class.php, 默认继承 Common 及 Action
 */
Class User {
    //控制器中默认的方法，用于获取用户默认的操作
    function index(){
        $user = D("users"); //模型 Users 类不存在，参数为表名
        $data=$user->select();//调用父类中的方法查询表中所有记录
    }
}

```





### ➤ 使用跨应用的 Model 类

如果项目中有前台和后台两个应用（也可以有更多的应用），是否需要各自定义一个业务模型对同一个表进行操作。例如，在后台应用(admin)中的 model 目录下声明一个 Users 类，类中声明了处理用户登录和退出的方法，如果在前台应用中的 model 目录下也声明一个 Users 类，在类中再写一次处理用户登录和退出的方法，就会发生代码重复编写的情况。所以在 BroPHP 框架中对同一个项目有多个应用时，相同表的处理可以使用同一个 Model 类来完成。当然也是使用系统内置的 D()函数完成，只不过除了使用第一个参数传一个类名外（或是表名），还需要使用第二个参数传另一个应用的应用目录名（和入口文件中声明的应用目录名同名）。例如，在前台应用的控制器 Index 类中的 index 方法中，使用后台应用（在 admin 目录下）中的模型 Users 类处理 users 表。D()函数的使用如下所示：

```
/*
 * 在前台 controls 目录下，文件名为 index.class.php
 */
Class Index {
    //控制器中默认的方法
    function index(){
        //创建后台 admin 目录中 models 目录下的 Users 类对象
        $user = D("users", "admin");
        $user->isLogin();//在前台调用后台 Users 类中的登录方法
    }
}
```

### ➤ 没有为 D()方法提供参数

如果在使用 D()方法时没有提供参数，则也可以创建 Model 类对象，但不能对数据表进行操作，只能完成一些非表操作的功能，例如获取数据库的使用大小、获取数据库系统的版本、事务处理等。D()函数的使用如下所示：

```
/*
 * 在前台 controls 目录下，文件名为 index.class.php
 */
Class Index {
    //控制器中默认的方法
    function index(){
        //如果没有传表名或类名，则直接创建 DB 对象，但不能对表进行操作
        $db = D(); //可以访问 DB 对象中非表的操作方法
        $db->dbSize(); //获取数据库的使用大小
        $db->dbVersion(); //获取数据库系统的版本
        $db->beginTransaction(); //开启事务
        $db->commit(); //提交事务
        $db->rollBack(); //事务回滚
    }
}
```

## 12.4 数据库的统一操作接口

BroPHP 框架中为所有对表的操作提供了统一的接口，这样不仅可以省去编写 SQL 语句的烦恼，也不用考虑 SQL 语句的执行效率和 SQL 优化以及 SQL 注入等安全问题，因为所有 SQL 语句





都在框架中已经封装好了。并且这些接口操作简单，符合程序员的开发习惯。在 BroPHP 框架中提供的数据库操作接口和描述如下表所示：

数据库的操作接口及描述

方法名	描述
insert()	向表中新增数据，返回最后插入的自动增长 ID
update()	更新表中的数据，返回更新的影响行数
delete()	删除表中的数据，返回删除的影响行数
field()	连贯操作时使用，设置查询的字段，返回对象\$this
where()	连贯操作时使用，设置查询条件，返回对象\$this
order()	连贯操作时使用，设置 SQL 的排序方式，返回对象\$this
limit()	连贯操作时使用，设置获取的记录数，返回对象\$this
group()	连贯操作时使用，设置 SQL 的分组条件，返回对象\$this
having()	连贯操作时使用，设置分组时的查询条件，返回对象\$this
total()	获取符合条件的记录总数
find()	获取数据表的单条记录，返回一维数组
select()	获取数据表的多条记录，返回二维数组
r_select()	关联查询，从有关联的多个表中获取数据
r_delete()	关联删除，一起删除多个表中的有关联的记录
query()	任意的 SQL 语句都可以使用该方法执行
beginTransaction()	开启事务处理操作
commit()	提交事务
rollback()	事务回滚
dbSize()	获取数据库使用大小
dbVersion()	获取数据库的版本
setMsg()	设置提示信息，该方法设置的消息可以通过 getMsg()方法获取
getMsg()	获取一些验证信息，提示给用户使用，一起可以获取多条，以字符串返回

以上的每个方法在使用时都不用提供表名，因为在使用这些方法时要先为数据表创建对应的 Model 类对象，而在使用 D()函数创建对象时已经传递了表名，也自动获取了表结构。每个方法的详细使用如下所示：

### ➤ insert([array \$post][, mixed filter][,bool validate])

该方法是向数据表中新增一条记录，只要为该函数提供正确的新增所需要的数据（是一个数组），就可以直接插入到表中。通常都是在控制器接收表单提交过来的数据，再在控制器中调用 Model 类中的这个方法完成添加数据。在向表中新增数据时需要注意以下两点：

1. 所以 form 表单的提交方法 method 最好使用 post 方式
2. 每个表单项的名称一定要和数据表的字段名相同，只有表单名和字段名相同的项才能加入到表中。

该函数有三个可选参数，如果没有提供第一个参数，则默认是将表单提交过来的数组 \$\_POST 作为第一个参数。也可以直接将 \$\_POST 数组作为第一个参数传递，当然也可以根据自己的需要组合一个数组后再传递给第一个参数。例如，bro\_users 表结构如下所示：



```

Create table bro_users(                                #表名为 bro_users
    id int not null auto_increment,                    #用户编号 ID
    name varchar(30) not null default '',              #用户名
    age int not null default 0,                        #用户年龄
    sex char(4) not null default '男',                 #用户性别
    ptime int not null default 0,                      #用户注册时间
    email varchar(60) not null default '',             #用户电子邮箱
    primary key(id)
);

```

例如，表单提交过来的数组\$\_POST，如下所示：

```

$_POST=array(
    "name"=>"admin",                                #<input name="name">
    "age"=>"22",                                     #<input name="age">
    "sex"=>"男",                                     #<input name="sex">
    "email"=>"gaolf@php.net",                        #<input name="email">
    "sub"=>"注册"                                    #<input name="sub" type="submit">
);

```

从\$\_POST 数组中可以看到表单中提交过来的数组，没有提交 id（表中是自动增长的）和 ptime（注册时间需要从 PHP 服务器自动获取）。而和 bro\_users 表字段不一样的还多一个名称为 sub 的提交按钮。在控制器的方法中使用如下所示：

```

/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中添加方法
    function add(){
        $user = D("users");                        //创建用户实例

        $_POST["ptime"]=time();                    //向$_POST 数组添加注册时间

        $id=$user->insert();                        //默认使用$_POST 数组作为参数
        //$id=$user->insert($_POST);                //也可以直接传递$_POST 参数
    }
}

```

按上例 insert()方法的使用，内部将组合成一个准备好的语句。如下所示：

```

$sql="INSERT INTO bro_users(name,age,sex,ptime,email) values(?,?,?,?);"
对应的数组绑定? 参数 array("admin", "22","男","123322122","gaolf@php.net");

```

在 insert()方法内部有一个处理，会将传递过来的\$\_POST 数组下标和表字段名进行匹配，如果有匹配成功的说明表单项的名称和数据表的字段名相同。例如“sub”=>“注册”下标“sub”就不是表的字段名，所以在组合 SQL 语句时将其去掉。

Insert()函数需要的第二个参数\$filter，默认值是 1（只要是“真”值都可以），这个参数决定是否对表单传递过来的数据进行过滤。因为表单是黑客攻击网站的主要入口，所以为了防止用户在表单中输出一些不充许的 HTML 标记或恶意的 JavaScript 代码，在 insert()中使用了 PHP



中内置的两个方法 `stripslashes()` 和 `htmlspecialchars()`，不仅能将 HTML 标记转为了 HTML 实体，同时也去掉了在表单中输入的单引号或双引号自动添加的转义符号。特定情况下可以使用 0 值 (只要是“假”值都可以) 参数关闭这个过滤功能，**如果使用一个数组作为参数，数组中的元素为表单名称，则也可以关闭部分。**

`Insert()` 函数也可以提供第三个参数 `$validata`，默认值是 0 (只要是“假”值都可以)，这个参数决定是否需要使用 XML 对数据进行自动验证，“假”值是不需要验证的。

`Insert()` 方法执行成功返回最后自动增长的 ID，失败返回 `false`，如果数据表没有自动增长的字段，成功返回 `true`。

## ➤ `update([array $array][, int filter] [, bool validata])`

该方法用于更新数据表中的记录，有三个可选参数，第二个和第三个参数和 `insert()` 方法中两个参数一样，用于设置表单过滤功能和设置自动验证。

该方法可以以主键为条件更新一条记录，也可以自己设置的条件同时更新多条记录，还可以设置更新特定的字段。例如，数据表 `bro_users` 的结构同上，`update()` 方法的常用的几种使用方式如下所示：

第一种：最常用 `update()` 方法更新一条数据，将修改表单提交过来的 `$_POST` 数组直接传给该函数的第一个参数 (不需要修改的字段可以在 `$_POST` 数组中去掉)，则会以 `$_POST` 数组中和表主键字段名称相同的元素下标，作为条件更新一条记录。例如，`$_POST` 数组中的内容如下：

```
$_POST=array(
    "id"=>"5",                #<input name="name" type="hidden">
    "name"=>"admin",          #<input name="name">
    "age"=>"25",              #<input name="age">
    "sex"=>"男",              #<input name="sex">
    "email"=>"gaolf@php.net", #<input name="email">
    "sub"=>"修改"             #<input name="sub" type="submit">
);
```

这里要注意修改表单的名称中一定要有一个对应表的主键 (本例是 ID，通常使用隐藏表单传递)，将这个 `$_POST` 作为每一个参数传入 `update()` 方法，使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中修改的方法
    function mod(){
        $user = D("users"); //创建用户实例
        $rows=$user->update(); //默认使用$_POST 数组作为参数
        //$rows=$user->update($_POST); //也可以直接传递$_POST 参数
    }
}
```

```
$sql="UPDATE bro_users SET name=?,age=?,sex=?,email=? WHERE id=?";
对应的数组绑定? 参数 array("admin", "25","男","gaolf@php.net",5);
```



第二种：可以通过 `where()` 方法（详见 `where()` 方法）使用连贯操作，设置更新的条件去更新一条或多条记录。例如，使用 `where()` 方法设置条件更新主键值为 1、2 和 3 的三条记录，使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中修改的方法
    function mod(){
        $user = D("users");           //创建用户实例
        //使用$_POST 数组作为参数（可以默认），加上 where() 连贯操作
        $rows=$user->where("1,2,3")->update($_POST);
    }
}
```

```
$sql="UPDATE bro_users SET name=?,age=?,sex=?,email=? WHERE id in(?,?,?)";
对应的数组绑定？参数 array("admin", "25","男","gaolf@php.net",1,2,3);
```

第三种：在前两种方式的基础上，还可以使用 `update()` 方法更新指定的字段。例如，计算一篇文章的访问数，访问一次访问数字段值就累加一次。本例设置 `bro_users` 表中 `id` 为 5 的记录中年龄字段(`age`)的值累加 1。也是使用 `update()` 方法的第一参数实现，只要在参数中使用一个字符串，这个字符串就是 SQL 语句中的 `SET` 后面的设置内容。使用和组合后的 SQL 语句如下：

```
$rows=$user->where(array("id">5))->update("age=age+1");
```

```
$sql="UPDATE bro_users SET age=age+1 WHERE id=?";
对应的数组绑定？参数 array(5);
```

`update()` 方法执行成功后，返回影响记录的行数（没有行数影响可以作为 `false` 值）。

## ➤ delete()

该方法用于删除数据表中的记录，可以以主键为条件删除一条记录，也可以自己设置的条件同时删除多条记录。其实 `delete()` 方法和 `where()` 方法（详见 `where()` 方法）的参数是一样的，可以任意设置条件删除记录。例如，数据表 `bro_users` 的结构同上，`delete()` 方法的常用的几种使用方式如下所示：

第一种：如果你想一条一条的单个删除记录，只要将主键（通常是 `id`）值作为参数传入即可。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中删除的方法
    function del(){
        $user = D("users");           //创建用户实例
        //使用$_GET 数组传过来的主键作为参数删除一条，例如 $_GET["id"]=5
        $rows=$user->delete($_GET["id"]);
    }
}
```

```
$sql="DELETE FROM bro_users WHERE id=?";
对应的数组绑定？参数 array(5);
```



第二种：通常在用户列表中可以通过复选框选中多条记录以后一起删除，只要将多记录的主键（像 id）组合成数组作为参数传入 `delete()` 方法即可。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中删除的方法
    function del(){
        $user = D("users");           //创建用户实例
        /*
         * $_POST 数组中是传过来的多个主键值(选中 id 为前 5 条)
         * 例如 $_POST=array("id">=>array(1,2,3,4,5));
         */
        $rows=$user->delete($_POST["id"]);
    }
}
```

```
$sql="DELETE FROM bro_users WHERE id IN(?,?,?,?,?)";
对应的数组绑定? 参数 array(1,2,3,4,5);
```

当然，`delete()` 方法也可以有其他用法，例如删除“id > 5”的所有记录，或是删除名称中包含“php”字符串的，也可以和 `where()` 组成连贯操作一起使用，总之条件可以任意设置。如下所示：

```
$rows=$user->delete(array("id ">=>5)); //删除 id > 5 的记录
或 $user->where(array("id ">=>5))->delete();
```

```
$sql="DELETE FROM bro_users WHERE id > ?";
对应的数组绑定? 参数 array(5);
```

```
$rows=$user->delete(array("name">=>"%php%")); //删除名称中包含 PHP 的记录
或 $user->where(array("name">=>"%php%"))->delete();
```

```
$sql="DELETE FROM bro_users WHERE name LIKE ?";
对应的数组绑定? 参数 array("%php%");
```

`delete()` 方法执行成功后，返回影响记录的行数（没有行数影响可以作为 false 值）。

## ➤ find()

该方法用于从一个数据表中获取满足条件的一条记录，以一维数组的方式返回查找到的结果。经常用在修改数据时先通过这个方法获取一条记录放到修改表单中。这个方法常用的使用方式有两种：

第一种：直接通过参数传入需要查找记录的主键（通常为 id），返回主键对应记录的一维数组，使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中修改的方法
    function mod(){
```



```

        $user = D("users");           //创建用户实例
        $data=$user->find($_GET["id"]); //$_GET["id"]=5
        P($data);                     //打印结果数组
    }
}

```

```

$sql="SELECT id,name,age,sex,email FROM bro_users WHERE id=? LIMIT 1";
对应的数组绑定? 参数 array(5);
结果数组: Array("id"=>5, "name"=>"zs", "age"=>20, "sex"=>"男", "email"=>"a@b.c");

```

第二种：可以通过 `where()` 方法（详见 `where()` 方法）和 `field()` 方法（详见 `field()` 方法）使用连贯操作，自己定义查询条件和查找指定的字段。例如，从 `bro_users` 表中查找一条 `id` 大于 5 的 `id`、`name` 和 `sex` 三个字段，通过 `field()` 方法设置查找 `id`、`name` 和 `sex` 三个字段，使用 `where()` 指定 `id` 大于 5 的查询条件。如下所示：

```

$data=$user->field("id,name,sex")->where(array("id ">"=5"))->find();

$sql="SELECT id,name,sex FROM bro_users WHERE id > ? LIMIT 1";
对应的数组绑定? 参数 array(5);

结果数组: Array("id"=>6, "name"=>"zs", "sex"=>"男");

```

## ➤ field()

该方法不能单独使用，需要和 `find()` 或 `select()` 方法一起使用，形成连贯操作去组合一个 SQL 语句，用于设置查询指定的字段。用法很简单，只要在 SQL 语句的“SELECT”和“表名”之间可以写的内容都可以写在这个方法的参数中。例如，和 `find()` 方法一起使用，如下所示：

```

$data=$user->field("id,name,sex")->find(5);
//SQL 语句如下:
$sql="SELECT id,name,sex FROM bro_users WHERE id = ? LIMIT 1";
对应的数组绑定? 参数 array(5);

```

或设置查找字段时为字段指定别名，如下所示：

```

$data=$user->field("id as '编号',name '用户名',sex '性别'")->find(5);
//SQL 语句如下:
$sql="SELECT id as '编号',name '用户名',sex '性别' FROM bro_users WHERE id = ? LIMIT 1";
对应的数组绑定? 参数 array(5);

```

## ➤ where()

该方法也不能单独使用，需要和 `find()`、`select()`、`update()`、`total()` 或 `delete()` 等方法之一一起使用，形成连贯操作去组合一个 SQL 语句，用于设置查询条件。例如，前面见过和 `find()`、`update()` 及 `delete()` 方法配合使用的方式。使用这个方法设置查询条件非常的灵活，有很多种使用方式，基本上可以通过这个方法组合成任意的查询条件，这个方法常用的使用方式如下：





第一种：如果没有传递参数，或条件为空（例如：”、0、false 等），则在 SQL 语句中不使用 where 条件。例如，从 bro\_users 表中使用 select() 获取数据，但组合条件 where 条件时没有传参，如下所示：

```
$data=$user->where('')->select(); //where('') 参数为空，或 0、false
//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users"; //没有 where 条件
```

第二种：如果直接在这个方法的参数中传一个整数，则组合的 where 条件就是直接设置主键（通常为自动增长的 id）的值。例如，从 bro\_users 表中查找 id（主键）为 5 的记录。如下所示：

```
$data=$user->where(5)->select(); //使用整数作为参数
//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE id=?";
对应的数组绑定？参数 array(5);
```

第三种：如果使用以逗号分隔的数字字符串或使用一维的索引数组作为参数，则组合的 SQL 语句是通过 IN 关键字为主键设置多个查询的值。例如，从 bro\_users 表中查找 id（主键）为 1,2,3 的三条记录。两种方式如下所示：

```
$data=$user->where('1,2,3')->select(); //使用数字字符串作为参数
$data=$user->where(array(1,2,3))->select(); //使用一维的索引数组作为参数

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE id IN(?,?,?)";
对应的数组绑定？参数 array(1,2,3);
```

第四种：如果是以一个关联数组作为参数，数组中的第一个元素还是一个数组（二维数组），则组合的 SQL 语句是通过 IN 关键字设置多个查询的值，元素下标作为字段名。例如，从 bro\_articles 表中查找 uid（非主键）为 1,2,3 的三条记录。两种方式如下所示：

```
$data=$user->where(array("uid"=>array(1,2,3)))->select(); //二维数组参数

//SQL 语句如下：
$sql="SELECT id,title,content FROM bro_articles WHERE uid IN(?,?,?)";
对应的数组绑定？参数 array(1,2,3);
```

第五种：如果是以一个关联数组作为参数，则数组的下标是数据表的字段名，数组的值是这个字段查询的值。例如，从 bro\_users 表中查找性别（sex）为“男”所有记录。如下所示：

```
$data=$user->where(array("sex"=>"男"))->select(); //使用关联数组作为参数

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE sex=?";
对应的数组绑定？参数 array("男");
```

第六种：如果还是以以一个关联数组作为参数，但在数组的值中使用两个百分号("%值%")，则会组合成模糊查询的形式。例如，从 bro\_users 表中查找名字（name）中包含字符串“feng”的所有记录。如下所示：

```
$data=$user->where(array("name"=>"%feng%"))->select(); //使用关联数组作为参数

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE name LIKE ?";
```





对应的数组绑定？参数 `array("%feng%");`

第七种：也是以一个关联数组作为参数，但关联数组的下标中使用“空格”分为两部分，空格前面是指定数据表的字段名，空格后面是指定的查询运算符。例如，从 `bro_users` 表中查找年龄（age）大于“20”岁的所有记录。如下所示：

```
$data=$user->where(array("age ">=>20))->select(); //使用关联数组作为参数

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE age > ?";
对应的数组绑定？参数 array(20);
```

第八种：如果参数的关联数组是由多个元素组成，则是设置多个 `where` 条件，多个条件之间使用“and”隔开，是“逻辑与”的关系。例如，从 `bro_users` 表中查找年龄（age）大于“20”岁，并且性别（sex）为“男”的所有记录。如下所示：

```
$data=$user->where(array("age ">=>20, "sex"=>"男"))->select(); //数组中多个元素

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE age >? AND sex=?";
对应的数组绑定？参数 array(20, "男");
```

第九种：如果参数是多个关联数组，则也是设置多个 `where` 条件，但多个条件之前是使用“or”隔开，是“逻辑或”的关系。例如，从 `bro_users` 表中查找名字（name）中包含字符串“feng”的，或者性别（sex）为“男”的所有记录。如下所示：

```
$data=$user->where(array("name"=>"%feng%"), array("sex"=>"男"))->select();

//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users WHERE name LIKE ? OR sex=?";
对应的数组绑定？参数 array("%feng%", "男");
```

第十种，也是最后一种：如果直接以字符串作为参数，就像直接写 SQL 语句中 `where` 条件一样。如果能通过前面几种方式完成 `Where` 条件设置就尽量不使用这种方式，因为这种方式不能使用“?”参数，也就不能防止 SQL 注入。例如，从 `bro_users` 表中查找年龄（age）大于“20”岁，并且性别（sex）为“男”的所有记录。如下所示：

```
$data=$user->where("age > 20 AND sex='男'"))->select(); //直接使用字符串参数

//SQL 语句如下：
"SELECT id,name,age,sex,email FROM bro_users WHERE age >20 AND sex='男'";
```

## ➤ order()

该方法也不能单独使用，需要和 `select()`、`delete()`、`update()` 方法及其他连贯操作的方法一起使用，用于设置 SQL 的排序条件，默认所有表都是按主键（通常为 `Id`）正序排序。如果需要改变查询结果的排序方式就可以通过这个方法实现。例如，从 `bro_users` 表中查找年龄（age）大于“20”岁的用户，并按年龄从大到小排序。如下所示：

```
$data=$user->where(array("age">20))->order("age desc")->select();

//SQL 语句如下：
"SELECT id,name,age,sex,email FROM bro_users WHERE age >20 ORDER age DESC";

//或删除年龄大于 20 岁的最后 5 条记录：
$row=$user->where(array("age">20))->order("age desc")->limit(5)->delete();
```



```
//SQL 语句如下：
"DELETE FROM bro_users WHERE age >20 ORDER age DESC Limit 5";
```

## ➤ limit()

该方法也不能单独使用，需要和 select()、delete()、update()方法及其他连贯操作的方法一起使用，用于 SQL 语句限制查询记录的个数。可以使用的方式有两种：

第一种：直接使用一个整数作为参数，就是限制记录的个数，如下所示：

```
$data=$user->limit(10)->select(); //取 10 条记录
```

```
//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users LIMIT 10";
```

第二种：可以使用两个整数作为参数（也可以以逗号分隔开两个数字的字符串作为参数），分别设置从那条记录开始查询和取多少条记录，如下所示：

```
$data=$user->limit(30,10)->select(); //从 30 条开始取，取 10 条记录，两个数字参数
$data=$user->limit('30,10')->select(); //从 30 条开始取，取 10 条记录，字符串参数
```

```
//SQL 语句如下：
$sql="SELECT id,name,age,sex,email FROM bro_users LIMIT 30,10";
```

第三种：将 limit()和更新 update()方法或删除 delete()方法一起使用。例如，将最新添加的 5 条记录的年龄加 1，如下所示：

```
//和 order() 及 update() 方法一起使用
$row=$user->Order("id desc")->limit(5)->update("age=age+1");
//SQL 语句如下：
$sql="UPDATE bro_users set age=age+1 ORDER BY id DESC LIMIT 5";
```

## ➤ group()

该方法也不能单独使用，需要和 select()方法及其他连贯操作的方法一起使用，用于对数据的查询记录设置分组条件。例如，在 bro\_users 表中按性别(sex)统计男生和女生两组的总记录数。如下所示：

```
$data=$user->field('sex, count(sex)')->group('sex')->select(); //按性别分组
```

```
//SQL 语句如下：
$sql="SELECT sex, count(sex) FROM bro_users GROUP BY sex";
```

## ➤ having()

该方法也不能单独使用，需要和 select()方法及其他连贯操作的方法一起使用，用于设置分组后的筛选条件设置，必须和 group()方法一起使用。例如，统计 bro\_users 表中平均年龄大于 20 岁的男生和女生数量。如下所示：

```
$user->field('sex, count(sex)')->group('sex')->having('avg(age)>20')->select();
```

```
//SQL 语句如下：
$sql="SELECT sex, count(sex) FROM bro_users GROUP BY sex HAVING avg(age)>20";
```



## ➤ total()

获取满足条件的记录总数，通常用于计算分页。可以和 `where()` 方法连贯操作设置条件，也可以直接在参数中传递查询条件。如果没有指定参数则获取表中所有记录总数。例如，统计 `bro_users` 表年龄(age)大于 20 的数量。如下所示：

```
$count=$user->total(array("age >"=>20)); //直接使用
$count=$user->where(array("age >"=>20))->total(); //和 where() 方法一起使用

//SQL 语句如下：
$sql="SELECT COUNT(*) as count FROM bro_users WHERE age > ?";
对应的数组绑定? 参数 array(20);
```

## ➤ select()

从一个数据表中获取满足条件的一条或多条记录，返回二维数组。具体的连贯操作参考前面的 `field()`、`where()`、`order()`、`limit()`、`group()`、`having()` 等方法。例如，从表 `bro_users` 中获取主键值为 1,2,3 的三条记录。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function index(){
        $user = D("users"); //创建用户实例
        $data=$user->field('id,name, age') //设置查询字段
            ->where('1,2,3') //设置查询条件
            ->order('id desc') //设置排序条件
            ->select(); //获取满足条件记录
        P($data); //打印二维数组
    }
}
```

```
$sql="SELECT id,name,age FROM bro_users WHERE id in(?,?,?) ORDER id desc";
对应的数组绑定? 参数 array(1,2,3);
```

返回的二维数组 `$data` 的格式：

```
$data=array(
    [0]=>Array("id"=>3, "name"=>"wangwu", "age"=>30),
    [1]=>Array("id"=>2, "name"=>"lisi", "age"=>20),
    [2]=>Array("id"=>1, "name"=>"zhangsan", "age"=>10)
);
```

## ➤ r\_select()

目前使用的数据库系统都是关联数据库系统，关联关系则是指表与表之间存在一定的关联关系（在一个表中使用外键保存另一个表的主键），通常我们所说的关联关系包括下面三种：

1. 一对一关联（1: 1）：一个用户一个购物车（用户表一条记录和购物车中一条记录关系）
2. 一对多关联（1: n）：一个类别中有多篇文章（类别表一条记录和文章表中多条记录关系）



### 3. 多对多关联 (n:m)：一个班级有多个学生，一个学生上多个班组的课

`r_select()`方法用于关联查询，可以按关联关系从多个表中获取记录。该方法和 `select()`一样可以通过连贯操作获取指定的记录。这个方法的参数需要传递一个或多个数组，每个数组关联一个数据表。例如，需要和其他两个数据表进行关联查询，则需要一起传递两个数组，每个参数的数组的结构都是一样的，数组中每个元素的作用如下：

第一个元素：是需要关联的表名

第二个元素：关联表的字段列表，如果使用 1 对 1 关联数组的方式（没有提供第四个元素时），关联的数据表字段名和主表的字段名是不能相同的（如果相同则从表和主表重名的字段将自动加上表名前缀 `user_name`, `user` 为表名，`name` 为重名字段），就需要在这个元素中，为和主表同名字段起个别名。在这个参数中使用空字符串或 `null`，则获取所有字段。

第三个元素：关联的外键

第四个元素：这个元素可以是一个数组或一个字段名称字符串，是可选的。如果没有提供这个参数，则是以 1 对 1 的表关系返回记录列表（右关联）。如果提供了第四个元素，是一个字段名称字符串，则是自己指定主表中某个字段需要和关系的表外键关联的键（也是以 1:1 的关联，但记录以主表为主，是左关联）；当设置这个参数为一个数组时，就会以子数组形式进行关联查询（适合 1 对多的表关系）。在这个数组中有四个可用的元素，分别介绍如下：

元素一：为子数组的下标

元素二：是子数组记录的排序方式（可选）

元素三：是限制子数组记录个数（可选）

元素四：是子数组查询的 `where` 条件（可选）

例如：有 `bro_cats`(类别表)、`bro_articles`(文章表)、`bro_test`（测试表）三个表，表结构和记录内容如下所示：

```
# bro_cats(类别表):
Create table bro_cats(                                #表名为 bro_cats
    id int not null auto_increment,                    #类别编号 ID
    name varchar(60) not null default '',              #类别名称
    desn text not null default '',                     #类别描述
    primary key(id)
);
```

在表中插入有 3 条记录，如下所示：

```
INSERT INTO bro_cats(name, desn) values('php','php demo');
INSERT INTO bro_cats(name, desn) values('jsp','jsp demo');
INSERT INTO bro_cats(name, desn) values('asp','asp demo');
```

```
# bro_articles(文章表):
Create table bro_articles(                             #表名为 bro_articles
    id int not null auto_increment,                    #类别编号 ID
    cid int not null default 0,                        #关联 cats 表的外键
    name varchar(60) not null default '',              #文章名称
    content text not null default '',                  #文章内容
    primary key(id)
```



```
);
```

在表中插入有 5 条记录，如下所示：

```
INSERT INTO bro_articles(cid, name, content) //php 类中 cid=1
values(1,'this article of php1', 'php content1');
INSERT INTO bro_articles(cid, name, content) //php 类中 cid=1
values(1,'this article of php2', 'php content2');
INSERT INTO bro_articles(cid, name, content) //jsp 类中 cid=2
values(2,'this article of jsp', 'jsp content');
INSERT INTO bro_articles(cid, name, content) //asp 类中 cid=3
values(3,'this article of asp1', 'asp content1');
INSERT INTO bro_articles(cid, name, content) //asp 类中 cid=3
values(3,'this article of asp2', 'asp content2');
```

```
# bro_tests(测试表):
Create table bro_tests(                                #表名为 bro_users
    id int not null auto_increment,                    #类别编号 ID
    cid int not null default 0,                        #关联 cats 表的外键
    test varchar(60) not null default '',              #测试字段
    primary key(id)
);
```

在表中插入有 4 条记录，如下所示：

```
INSERT INTO bro_tests(cid, test) values(1,'php data'); //cid=1
INSERT INTO bro_tests(cid, test) values(2,'jsp data'); //cid=2
INSERT INTO bro_tests(cid, test) values(3,'asp data'); //cid=3
```

例如，使用 `r_select()` 方法从 `bro_cats` 和 `bro_articles` 两个表中获取类别名称、文章名称和文章内容。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class Cat {
    //控制器中默认方法
    function index(){
        $cat=D("cats");
        $data=$cat->field('id,name as cname') //主键 id 必取
            ->r_select(
                //数组 关联的表名 字段列表 外键
                array('articles', 'id,name,content', 'cid'));

        p($data); //打印二维数组
    }
}
```

```
SELECT id,name as cname FROM bro_cats ORDER BY id ASC
SELECT id,name,content,cid FROM bro_articles WHERE cid IN('1','2','3') ORDER BY id ASC
```

返回的二维数组 `$data` 的格式：

```
$data=Array (
    [0] => Array(
```



```
[id] => 1
[cname] => php
[name] => this article of php1
[content] => php content1
[cid] => 1
)
[1] => Array (
    [id] => 1
    [cname] => php
    [name] => this article of php2
    [content] => php content2
    [cid] => 1
)
[2] => Array (
    [id] => 2
    [cname] => jsp
    [name] => this article of jsp
    [content] => jsp content
    [cid] => 2
)
[3] => Array (
    [id] => 3
    [cname] => asp
    [name] => this article of asp1
    [content] => asp content1
    [cid] => 3
)
[4] => Array (
    [id] => 3
    [cname] => asp
    [name] => this article of asp2
    [content] => asp content2
    [cid] => 3
)
)
```

例如，还是使用 `r_select()` 方法从 `bro_cats` 和 `bro_articles` 两个表中获取类别名称、文章名称和文章内容，但要求让 `bro_articles` 表中的记录以子数组的形式和 `bro_cats` 记录对应显示，这时，就需要在参数的数组中使用第 4 个元素，这个元素可以是一个数组（有 4 个可以用的元素）。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class Cat {
    //控制器中默认方法
    function index(){
```



```

        $cat=D("cats");
        $data=$cat->field('id,name') //不需要别名
            ->r_select(
            array('articles', 'id,name,content', 'cid',
                array('art', "id desc",'5')));
        p($data); //打印二维数组
    }
}

```

```

SELECT id,name as cname FROM bro_cats ORDER BY id ASC
SELECT id,name,content,cid FROM bro_articles WHERE cid IN('1','2','3') ORDER BY id ASC

```

返回的二维数组\$data 的格式:

```

$data= Array (
    [0] => Array(
        [id] => 1
        [name] => php
        [art] => Array( //子数给下标 art
            [0] => Array(
                [id] => 2 //order by id desc
                [name] => this article of php2
                [content] => php content2
                [cid] => 1
            )

            [1] => Array(
                [id] => 1
                [name] => this article of php1
                [content] => php content1
                [cid] => 1
            )

        )
    )

    [1] => Array (
        [id] => 2
        [name] => jsp
        [art] => Array(
            [0] => Array(
                [id] => 3
                [name] => this article of jsp
                [content] => jsp content
                [cid] => 2
            )

        )
    )
)

```





```

[2] => Array (
    [id] => 3
    [name] => asp
    [art] => Array (
        [0] => Array (
            [id] => 5
            [name] => this article of asp2
            [content] => asp content2
            [cid] => 3
        )

        [1] => Array (
            [id] => 4
            [name] => this article of asp1
            [content] => asp content1
            [cid] => 3
        )
    )
)

```

如果从三个关联的表中获取数据(加上 `bro_tests` 表,关联的外键都是 `cid`)，只要在 `r_select()` 方法中多传入一个数组即可（可以更多个关联的表）。使用和组合后的 SQL 语句如下：使用和组合后的 SQL 语句如下：

```

/*
 * 定义一个控制器类 cat
 */
Class Cat {
    //控制器中默认方法
    function index(){
        $cat=D("cats");
        $data=$cat->field('id,name') //不需要别名
        ->r_select(
            array('articles', 'id,name,content', 'cid',
                array('art', "id desc"))
            array('tests', 'id,test ', 'cid',
                array('test') )
        );

        p($data); //打印二维数组
    }
}

```

```

SELECT id,name as cname FROM bro_cats ORDER BY id ASC
SELECT id,name,content,cid FROM bro_articles WHERE cid IN('1','2','3') ORDER BY id ASC

```

## ➤ r\_delete()



该方法用于关联查询，可以按关联关系从多个表中删除关联的数据记录。和关联查询相似，只要在这个方法的参数中传递一个或多个数组，每个数组对应一个关联的数据表，数组中有三个元素，第一个元素为关联的表名，第二个元素为关联的外键，**第三个元素是可选的附加条件，为一个数组，如果需要多个附加条件和 where 方法一样为这个数组提供多个元素即可。**例如，表结构同上，删除类别表 bro\_cats 中 id 为 1,2 两条记录，同时删除 bro\_articles 和 bro\_tests 中和类别对应的记录。使用和组合后的 SQL 语句如下：

```
/*
 * 定义一个控制器类 User
 */
Class Cat {
    //控制器中默认方法
    function index(){
        $cat=D("cats");
        $data=$cat->where("1,2")->r_delete(array("articles", "cid"), array("tests", "cid"), array("demos", "cid", array("type"=>1)));
        P($data); //返回影响行数
    }
}
```

```
DELETE FROM bro_articles WHERE cid IN('1','2')
DELETE FROM bro_tests WHERE cid IN('1','2')
DELETE FROM bro_demos WHERE id IN('1','2') AND 'type'='1'
DELETE FROM bro_cats WHERE id IN('1','2')
```

## ➤ query()

SQL 语句的统一入口，任何用户自定义的 SQL 语句(不能通过前面方法完成的 SQL 语句)，都可以通过这个方法完成。该方法有三个参数：第一个参数就是用户自定义 SQL 语句，是必选项，可以使用“？”参数，如果使用问号参数就必须在该方法的第三个参数中使用数组为“？”参数绑定对应的值。该方法的第二个参数是指定 SQL 语句的形式，返回什么类型由这个参数决定，第二个参数可以使用的字符串包括：

- select (查询多条记录的操作，返回二维数组)
- find (查询一条记录的操作，返回一维数组)
- total (按条件查询数据表的总记录数)
- insert (插入数据的操作，返回最后插入的 ID)
- update (更新数据表的操作，返回影响的行数)
- delete (删除数据表的操作，返回影响的行数)

如果第二个参数为空，或其他字符串，则 query() 方法执行成功返回 true，失败返回 false。使用的方式如下：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function index(){
        $user = D("users"); //创建用户实例
    }
}
```



```

        $total=$user->query("SELECT 内容任意 FROM
bro_users","total");
        P($total);                                //获取总数
        $data=$user->query("SELECT * FROM bro_users","select");
        P($data);                                //打印二维数组
    }
    function add(){
        $user = D("users");                        //创建用户实例
        //自定义 insert 语句， 使用?参数， 用最后一个数组参数绑定值
        $num=$user->query("insert into {$user->tabName} (name, age,
sex,email) values(?,?,?,?)", "insert", array('zhangsan',10, '男',
'aaa@bbb.com')); //返回最后插入的 ID
    }
    function del(){
        $user = D("users");                        //创建用户实例
        //自定义删除 SQL 语句， 删除 age > 20
        $num=$user->query("DELETE FROM {$user->tabName} WHERE age
> ?", "delete", array(20));                      //返回影响的行数
    }
    function mod(){
        $user = D("users");                        //创建用户实例
        //自己定义 update 语句， 更新 id=2 的数据
        $num=$user->query("UPDATE {$user->tabName} set
name=?,age=?,sex=?,email=? WHERE id=?", "update", array("zhangsan",
15, "女", "bbb@ccc.com", 2));                    //返回影响的行数
    }
    function create(){
        $user = D("users");                        //创建用户实例
        //自己定义创建表 hello 的语句， 在第二个参数使用空字符串
        $user->query("create table if not exists hello(id int, name
varchar(30)) ", "");                             //成功返回 true
    }
}

```

表名可以直接使用数据库对象获取（\$tabName），例如：\$tableName=\$user->tableName;

### ➤ beginTransaction()

用于事务处理，开启一个事务。

### ➤ commit()

用于事务处理，提交事务。

### ➤ rollback()

用于事务处理，回滚事务。

### ➤ dbSize()



获取项目中所有数据表的使用大小。

### ➤ dbVersion()

获取数据库的版本信息。

### ➤ setMsg()

设置 Model 类中的提示消息，有一个参数。参数的类型可以是一个字符串，也可以是一个数组。该函数设置的提示消息可以通过 getMsg() 方法获取。

### ➤ getMsg()

获取 Model 类中的提示消息。例如，验证成功或失败返回的提示消息。

## 13 视图 View

视图（View）是用户看到并与之交互的界面，对 Web 应用程序来说，HTML 在视图中扮演着重要的角色。View 层用于与用户的交互，Controller 层是 Model 与 View 之间沟通的桥梁，它可以分派用户的请求并选择恰当的视图以用于显示。BroPHP 框架内置最流行的 Smarty 模板引擎，所以所有的视图都是由 Smarty 编写的模板（参考 Smarty 语法编写）。向视图中分配动态数据并显示输出模板都在控制器类的某个操作方法中完成，而模板文件把接收到的数据转换成相应的数据格式显示。

我们自定义的控制器类都间接的继承了 Smarty 类，所在在每个控制器类中都可以直接使用 (\$this) 访问从 Smarty 类中继承过来的方法和属性。常用的 Smarty 方法如下所示：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function index(){
        //向模板中分配变量
        $this->assign("data", $data);
        //输出模板（这个方法在 BroPHP 框架中改写了）
        $this->display();
        //判断模板是否已经被缓存
        $this->is_cached();
        //清除单个模板缓存
        $this->clear_cache();
        //清除所有缓存的模板
        $this->clear_all_cache();
    }
}
```

另外，对 Smarty 源代码有一个点改动，主要有两处：一个就是对 Smarty 的插件修改器 truncat，改写可以用于截取 utf8 字符集；另一处，加了一个块函数插件 nocahce，在模板文件中，<{nocache}></nocache> 块中的数据不被缓存。



## 13.1 切换模板风格

所有的视图都要将模板声明在当前项目应用的 **views** 目录下。因为可以为同一个应用程序编写多套模板，所以在 **views** 目录下声明的每个目录都是为当前的应用创建的一套独立的模板风格，默认的风格声明在 **default** 目录下。如果为一个应用编写了几套风格模板，只要修改配置文件中的“**TPLSTYLE**”选项即可（选项值和目录名对应）。如下所示：

```
/* 修改配置文件 config.inc.php */
define("TPLSTYLE", "default"); //找 views/default/下面的模板风格显示
//define("TPLSTYLE", "home1"); //找 views/home1/下面的模板风格显示
//define("TPLSTYLE", "home2"); //找 views/home2/下面的模板风格显示
```

如果项目中有两个或多个应用，又都使用同一个配置文件。例如，项目分为前台和后台两个应用，如果在配置文件中将 **TPLSTYLE** 改变，则前后台都要有对应的模板。如果只想前台有多套模板风格，而后台只要一套默认的模板风格，就需要为每个应用创建一个子配置文件，将上例的选项“`define("TPLSTYLE", "default");`”写在子配置文件中，并在每个入口文件的最上面包括这个子配置文件即可，或将这个选项直接写在每个应用入口文件的最上面。

## 13.2 模板文件的声明规则

在每套模板目录下有两个默认的目录 **public** 和 **resource**，**public** 目录下声明的是当前风格的公用模板文件。例如，**header.tpl** 模板、**footer.tpl** 模板等，默认有一个 **success.tpl** 模板，用来显示提示消息框（**success()**和 **error()**两个方法中输出）。**resource** 目录是这套模板风格共用资源目录，包括模板中用到的 **css**、**js** 和 **image**。

在 **BroPHP** 框架中，对父类 **Smarty** 中的 **display()** 方法重写改写过，所以声明模板的位置和模板文件名要按一定的规则。一个项目应用通常都会有多个模块（一个模块一个控制器类），需要在对应的风格模板目录下，为每个模块单独创建一个目录（目录名和控制类名相同，但全小写）。然后，在这个目录下创建和控制器中的操作方法同名的模板文件，模板文件的后缀名由配置文件 **config.inc.php** 中的“**TPLPREFIX**”选择决定，默认是“.tpl”，可以修改为.html 或.htm 以及其他的后缀名。

## 13.3 display()用新用法

**display()** 方法重载了父类 **Smarty** 中的方法，其他的参数都没有变量，只是将第一参数的用法改写了。如下所示：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function index(){
        /* 如果没有提供参数，默认找和当前模块相同目录名(user)下的
```



```

        * 默认模板文件名为当前操作名 (index), 后缀名为 tpl (配置文件改)
        * 例如: view/default/user/index.tpl 模板文件
        */
$this->display();

/* 如果提供参数没有"/", 默认找和当前模块相同目录名 (user) 下的
* 模板文件名为参数名 add, 后缀名为 tpl (配置文件改)
* 例如: view/default/user/add.tpl 模板文件
*/
$this->display("add");

/* 如果提供参数有"/", 找和 "/" 前模块同名目录名 (shop) 下的
* 模板文件名为参数名 add, 后缀名为 tpl (配置文件改)
* 例如: view/default/shop/add.tpl 模板文件
*/
$this->display("shop/add");
    }
}

```

## 13.4 在模板中可以直接使用的几个常用变量

在编写模板文件时，经常会用到链接地址、CSS 文件的地址或是 js 文件的地址，如果直接写 URL，那么当域名或主入口文件有改变，则所有 URL 都需要重新修改，所以在控制器的操作中自动将一些常用的 URL（和服务器对应）自动分配到了模板中，可以在模板中直接使用。

```

/* 例如，在 add.tpl 模板中（项目声明在 shop 目录下，入口文件为 admin.php，模块为 index） */
<{$root}>;           //到项目应用的根目录           /shop
<{$app}>;            //到项目应用的主入口文件        /shop/admin.php
<{$url}>;            //到访问的模块                  /shop/admin.php/index
<{$public}>;         //所有就用的共用资源的 public   /shop/public
<{$res}>;            //到模板风格下的 resource 目录   /shop/views/default/resource

例如：
<a href="<{$url}>/mod/id/5">修改</a>
<script src="<{$res}>/js/jquery.js"></script>

```

## 13.5 在服务器中可以直接使用的几个常用变量

虽然 BroPHP 框架对所有的类库和函数都是自动包含的，但如果需要在控制器或模型中加载自定义加 PHP 某个文件，可以通过 PROJECT\_PATH 和 APP\_PATH 两个路径完成。如下所示：

- **PROJECT\_PATH**     //代表项目所在的根路径，即与框架所在的目录同级
- **APP\_PATH**        //代表项目中当前应用目录（在入口文件中指定的路径）

另外，除了在模板文件中可以直接使用<{\$root}>、<{\$app}>、<{\$url}>、<{\$public}>、<{\$res}>等路径。如果不是使用模板文件，而是在 PHP 中直接去访问前台文件（js、CSS、HTML、图片等），则可以使用 BroPHP 框架中的几个常量或几个全局\$GLOBALS 变量，都是从 WEB 服务器根目录开始的绝对路径。如下所示：

- **B\_ROOT** 或 **\$GLOBALS["root"]**     //Web 服务器根到项目的根





- **B\_APP** 或 `$GLOBALS["app"]` //当前应用脚本文件
- **B\_URL** 或 `$GLOBALS["url"]` //访问到当前模块
- **B\_PUBLIC** 或 `$GLOBALS["public"]` //项目的全局资源目录
- **B\_URL** 或 `$GLOBALS["res"]` //当前应用模板的资源

还有，就是可以通过`$_GET["m"]`获取当前访问的模块名，也可通过`$_GET["a"]`访问当前的操作名称。

## 14 自动验证

自动验证是基于 XML 方式实现的，可以对所有表单在服务器端通过 PHP 实现自动验证。如果自己定义一个 js 文件，通过处理 XML 文件也可以同时实现在前台也自动使用 JavaScript 验证。使用方法是在当前应用的 models 目录下，创建一个和表名同名的 XML 文件。例如，对 bro\_users 表进行自动验证，则在 models 下创建一个 users.xml 文件(一般都是对入库的数据进行验证，而入库又发生在添加或修改数据时，所以 XML 文件名必须和表名相同才能自动处理)。文件中的使用样例如下所示：

```
/* 在 models 目录下，users.xml，对添加或修改 bro_users 表的表单进行自动验证 */
users.xml 文件
<?xml version="1.0" encoding="utf-8"?>
<form>
    <input name="name" type="notnull" action="both" msg="有问题" />
    <input name="email" type="email" msg="不是正确的 EMAIL 格式" />
    <input name="price" type="currency" msg="价格必须是金钱格式" />
    <input name="code" type="vcode" msg="验证码输入错误!" />
    <input name="name" type="regex" value="/^abc/i" msg="不能匹配!" />
</form>
```

在上例的 XML 文件中，最外层标记`<form>`和每个子标记`<input>`其实是可以任意命名（上例的命名类似表单），如果不是正确的 XML 文件格式，也会在调试模式下提示。但每个`<input>`标记中的属性名必按规范设置，也可以对同一个表单进行多次不同形式的验证（例如，年龄不能为空和年龄必须是整数等），只要连续写几个`<input>`标记即可。属性的设置分别介绍如下所示：

### ➤ name 属性

该属性是必须的属性，和提交的表单项 name 属性是对应的，表示对那个表单项进行验证。

### ➤ action 属性

该属性是可选的，用于设置验证的时间，可以有三个值 add(添加数据时进行验证)、mod(修改数据时进行验证)、both(添加和修改数据时都进行验证)。如果不加这个属性默认值是 both。

### ➤ msg 属性



该属性也是必须提供的属性，用于在验证没通过时的提示消息。

### ➤ value 属性

该属性也是可选的，不过该属性是否使用和设置的值都由 **type** 属性的值决定

### ➤ type 属性

这是一个可选的属性，用于设置验证的形式，如果没有提供这个属性，默认值是“**regex**”（使用正则表达式进行验证，需要在 **value** 的属性中给出正则表达式）。该属性可以使用的值及使用如下所示：

**regex**：使用正则进行验证，需要和 **value** 属性一起使用，在 **value** 中给出自定义的正则表达式，这也是默认的方式。例如：

```
<input name="name" type="regex" value="/^php/i" msg="名字不是以 PHP 开始！" />
```

**unique**：唯一性校验，检查提交过来的值，在数据表是否已经存在，例如：

```
<input name="name" type="unique" msg="这个用户名已经存在！" />
```

**notnull**：验证表单提交的内容是否为空。例如，只在添加数据时验证：

```
<input name="name" type="notnull" action="add" msg="用户名不能为空！" />
```

**email**：验证是否是正确的电子邮件格式。例如：

```
<input name="email" type="email" msg="不是正确的 EMAIL 格式！" />
```

**url**：验证是否是正确的 URL 格式。例如：

```
<input name="url" type="url" msg="不是正解的 URL 格式！" />
```

**number**：验证是否是数字格式。例如：

```
<input name="age" type="number" msg="年龄必须输出数字！" />
```

**currency**：验证是否为金钱格式。例如：

```
<input name="price" type="currency" msg="商品价格的录入格式不正确！" />
```

**confirm**：检查两次输入的密码是否一致，需要使用 **value** 属性指定另一个表单（第一个密码字段）名称。例如：

```
<input name="repassword" type="confirm" value="password" msg="两次密码输入不一致！" />
```

**in**：检查值是否在指定范围之内，需要使用 **value** 属性指定范围，有多种用法。例如：

```
<input name="num" type="in" value="2" msg="输出的值必须是 2！" />
<input name="num" type="in" value="2-9" msg="输出的值必须在 2 和 9 之间！" />
```



```
<input name="num" type="in" value="1, 3, 5, 7" msg="必须是 1,3,5,7 中的一个！" />
```

length: 检查值的长度是否在指定的范围之内，需要使用 value 属性指定范围，例如：

```
<input name="username" type="length" value="3" msg="用户名的长度必须为 3 个字节！" />
<input name="username" type="length" value="3," msg="用户名的长度必须在 3 个以上！" />
<input name="username" type="length" value="3-" msg="用户名的长度必须在 3 个以上！" />
<input name="username" type="length" value="3,20" msg="用户名的长度必须在 3-20 之间！" />
<input name="username" type="length" value="3-20" msg="用户名的长度必须在 3-20 之间！" />
```

callback：使用自定义的函数，通过回调的方式验证表单，需要通过 value 属性指定回调函数的名称。例如，使用自定义的函数 myfun 验证用户名，如下所示：

```
<input name="name" type="callback" value="myfun" msg="名字不是以 PHP 开始！" />
```

另外，如果使用 BroPHP 中提供的 Vcode 类输出验证码，只要表单中输入验证的选项名称 name 值为“code”，并且 XML 文件存在，就会自动验证。

**注意：**在使用框架中的添加和修改方法时，必须使用第三个参数开启自动验证

```
D("user")->insert($_POST, 1, 1); //第三个参数为真值开启自动验证
D("user")->update($_POST, 1, 1); //第三个参数为真值开启自动验证
```

可以使用 DB 中的 getMsg()方法获取 XML 标中的 msg 值，提示用户定义的错误报告。

## 15 缓存设置

BroPHP 提供了两种缓存机制，可以同时使用。一种是基于 memcached 将 session 会话数据和数据表的结果集缓存在服务器的内存中；另一种是使用 Smarty 的缓存机制静态化页面。

### ➤ 基于 memcached 缓存设置

BroPHP 框架的 memcached 缓存设置比容易，只要 memcached 服务器安装成功（可以有多台），并为 PHP 安装好了 memcached 的扩展应用。在配置文件 config.inc.php 中设置一个或多个 memecache 服务器地址和端口即可。BroPHP 框架就会自动将 session 信息和从数据库获取的结果集缓存到 memcached 中，如果用户执行了添加、修改或删除等有影响表行数的操作，则会重新将数据表的结果数据缓存。配置文件中启用 memcached 如下所示：

```
//使用单一 memcached 服务器
$memServers = array("localhost", 11211);

//如果有多台 memcache 服务器可以使用二维数组
$memServers = array(
    array("www.lampbrother.net", '11211'),
    array("www.brophp.com", '11211'),
    ...
);
```

另外，如果 BroPHP 框架的多个项目使用同一个 memcached 服务器时，实现了独立缓存，不会发生冲突。



## ➤ 基于 Smarty 的缓存机制

这种缓存设置和 Smarty 的使用方式是完全一样的，在 BroPHP 框架中也是通过配置文件 config.inc.php 去设置缓存（建议在开发时关闭缓存，上线运行后开启缓存）。

```
//在配置文件 config.inc.php 中开启 smarty 缓存设置
define("CSTART", 1);           //缓存开关 1 开启，0 为关闭
define("CTIME", 60*60*24*7);    //设置缓存时间
```

除了开启了缓存设置以外，还需要在控制器类中有些设置，同 Smarty 的用户一样，如下所示：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function index(){
        //如果有缓存则不再去连接数据库和执行 SQL 查询
        if(!$this->is_cached(null, $_SERVER["REQUEST_URI"])){
            //连接了数据库，读取表的数据
            $user=D("users");
            $this->assign("data", $user->select());
        }

        $this->display(null, $_SERVER["REQUEST_URI"]);
    }
}
```

也可以使用<{nocache}></nocache>设置局部不去缓存，这个 Smarty 块函数插件在 BroPHP 框架中已经设置好了，可以直接使用。

## 16 调试模式

调试模式是为程序员在开发阶段提供的帮助功能，在项目上线运行后将其关闭即可。关闭和开启调试模式非常简单，只要在配置文件 config.inc.php 中设置“DEBUG”选项的值即可（上线后使用 0 值关闭，开发时使用 1 值开启）。如果在线运行后，关闭了调试模式则会将运行中产生的异常写到 runtime 目录下的 error\_log 文件中，这样在运行后也可以通过查看这个文件对项目进行维护。调试模式中可供参考的信息包括：脚本运行时间、自动包含的类、运行中的异常、一些常见的提示、使用的 SQL 语句和表结构等，可以通过关闭按钮临时关闭掉输出的提示框。调试框的界面如下所示：



运行信息(0.0296 秒):

关闭X

```
【自动包含】
MemcacheModel 类
MemSession 类
Structure 类
Prouri 类
Smarty 类
MyTpl 类
Action 类
Common 类
IndexAction 类
【系统信息】
没有开启页面缓存! (但可以使用)
启用了Memcache
开启会话Session (使用Memcache保存会话信息)
当前访问的控制器类在项目应用目录下的: ./controls/index.class.php 文件!
```

如果在开发阶段，某个操作的视图中不想使用调试模式中的输出，则可以在操作中加上一个开关（使用函数 `debug(0)` 或 `debug()`，也可以使用 `$GLOBALS["debug"]=0`）就不会输出提示界面了。如下所示：

```
/*
 * 定义一个控制器类 Index
 */
Class Index {
    //控制器中默认方法
    function index() {
        //关闭调试模式的输出,或$GLOBALS["debug"]=0;
        debug(0);
    }
}
```

## 17 内置扩展类库

BroPHP 内置了几个常用的扩展类，不用需要任何改动直接就可以使用，包括文件上传、图像处理、分页和验证码 4 个类。分别介绍如下所示：

### 17.1 分页类 Page

分页功能在每个项目中都是很常见，该类的构造方法中有四个参数：

第一个参数是必须的，提供数据表需要显示的总记录数；

第二个参数是可选的，提供每页需要显示的记录总数，默认为 25 条；

第三个参数也是可选的，用来向下个页面提供本页中的数据。

第四个参数也是可选的，用来设置默认页，需要一个布尔值，默认为 `true`，如果为 `true` 值则默认显示第一页，如果使用 `false` 值则默认页为最后一页。

使用的方式如下所示：

```
/*
 * 定义一个控制器类 User
 */
```



```

Class User {
    //控制器中默认方法
    function index(){
        //创建用户对象
        $user=D("users");
        //创建分页对象， 第页显示 5 条数据
        $page=new Page($user->total(), 5);
        //获取每页数据
        $data=$user->limit($page->limit)->select();
        //将数据分配给模板
        $this->assign("data", $data);
        //分配分页内容给模板
        $this->assign("fpage", $page->fpage());
        //显示输出模板
        $this->display();
    }
}

```

输出结果如下所示：

共 33 条记录 本页 5 条 本页从 16-20 条 4/7页 [首页](#) [上一页](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [下一页](#) [末页](#)

### ➤ 设置输出形式

如果想自定义输出分页信息，也可以通过分页对象中的 `set()` 方法连贯操作进行设置，可以设置一个也可以单独或连续设置多个（设置的值都可以使用图片）。使用方式如下：

```

$page->set("head", "条图片")
    ->set("first", "|<")
    ->set("last", ">|")
    ->set("prev", "|<<")
    ->set("next", ">>|");

```

输出结果如下所示：

共 33 条图片 本页 5 条 本页从 16-20 条 4/7页 [⏪](#) [⏩](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [⏴](#) [⏵](#)

### ➤ 设置输出内容

如果只需要在输出结果中显示自己定义内容，也可以通过 `Page` 类中的 `fpage()` 方法的参数指定。在输出的结果中共有 8 部分给成，可以通过在 `fpage()` 的参数中传入“0-7”之间的整数自定义输出内容和输出的顺序。`fpage()` 的方法使用如下所示：

```

$this->assign("fpage", $page->fpage(4,5,6,0,3));

```

输出结果如下所示：

[⏪](#) [⏩](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [⏴](#) [⏵](#) 共 33 条图片 4/7页

### ➤ 附加资源

如果从当前页需要转到下一页时，将本页的一些数据也带到下一个页面中去，就可以在创建 `Page` 对象时，通过设置第三个参数完成。例如，当前是分类 `cid=5` 下面的数据分页，转到下



页时也要是 cid=5 类别下的数据。创建分页对象如下所示（可以传更多的数据过去，只要使用 PATHINFO 的格式）：

```
$page=new Page($total, NUM, "cid/5");
```

#### ➤ 可以获取的属性

可以从分页对象中获取两个属性的值，一个是分页使用的 limit，另一个则是当前的正在访问的页面：

```
$page->limit;    //用于 SQL 语句中
$page->page;     //获取当前的分页数
```

## 17.2 验证码类 Vcode

验证码也是在每个项目中都是很常见，用于限制“人”操作而非机器，该类的构造方法中有三个参数：第一个参数是验证码图片的宽度，默认值是 80 像素；第二个参数是验证码图片的高度，默认值是 20 像素；第三个参数是设置验证码的个数，默认值是 4 个。使用非常简单，只要在控制中声明一个方法，并在这个方法中创建对象后直接输出，然后在表单中使用<img>的 src 指定这个操作方法即可输出验证码。（注意：表单 name 名称为“code”）如下所法：

```
/*
 * 定义一个控制器类 User
 */
Class User {
    //控制器中默认方法
    function code(){
        //直接输出验证码对象
        echo new Vcode();
        //或 echo new Vcode(100, 25, 5); 使用参数设置验证码
        //或 echo new Vcode(120, 25, 6); 使用参数设置验证码
    }
}
```

在表单中使用方法

```
//在表单中使用验证码， <{$url}>/code 访问上例的 code() 操作，获取图片
<input type="text" name="code" > 
```

如果看不清可以点击图片换一张，或让用户感觉不区分大小写，可以使用 JavaScript 完成。如下所示：

```
//在表单中使用验证码，加上 js 事件
<input type="text" name="code" onkeyup="if (this.value != this.value.toUpperCase())
this.value=this.value.toUpperCase();" >
```





```
/code/'+Math.random()">
```

输出结果：



如果使用 BroPHP 自动验证，则只要 XML 文件存在，则会自动检查验证码（输出表单名称必须为“code”，因为在服务器中是使用\$\_SESSION[“code”]保存的验证码，并且在自动验证中也是使用 code 调用的函数）

## 17.3 图像处理类 Image

在项目开发时经常需要对上传的图片内容进行优化，最常见的操作是对图片进行缩放和加水印，本类提供了这两个功能。和前两个类的用法相似，只要创建对象后调用 thumb() 方法对图片进行缩放，而调用 waterMark() 方法可以为图片加水印（目前支持：gif, jpeg, png 等图片格式）。如下所示：

### ➤ 构造方法

该方法用来创建图像对象，参数是可选的，用来指定处理图片的位置，默认处理图片的目录是与框架在同级目录下的 public/uploads/ 目录下，可以自己定义目录位置。

### ➤ thumb() 方法

该方法用来对图像进行缩放，需要 4 个参数，其中最后一个参数是可选的，缩放成功后返回图片的名称，如果缩放失败则返回 false。参数说明如下：

- 第一个参数：是需要处理的图片名称（图片所在位置由构造方法决定）
- 第二个参数：图片需要缩放的宽度
- 第三个参数：图片需要缩放的高度
- 第四个参数：是可选的，指定缩放后图片新名的前缀，默认值为“th\_”

```
//例如，创建图像类对象后，将图片 brophp.gif 缩放至 300 x 3000，加上 th_前缀名
$img=new Image(); //创建图片对象
$imgname=$img->thumb("brophp.gif", 300, 300, "th_"); //缩放图片
```

### ➤ waterMark();

该方法用来为图像添加水印（只支持图片水印），也需要 4 个参数，其中最后一个参数也是可选的，成功后返回新图片的名称，如果失败则返回 false。参数说明如下：

- 第一个参数：背景图片，即需要加水印的图片（图片所在位置也由构造方法决定）
- 第二个参数：图片水印，即作为水印的图片（图片所在位置也由构造方法决定）
- 第三个参数：水印图片在背景图片上添加的位置，共有 10 种状态，0 为随机位置
  - 1 为顶端居左，2 为顶端居中，3 为顶端居右；
  - 4 为中部居左，5 为中部居中，6 为中部居右；
  - 7 为底端居左，8 为底端居中，9 为底端居右；
- 第四个参数：是可选的，指图片新名的前缀，默认值为“wa\_”

```
//例如，创建图像类对象后，将图片 brophp.gif 加上水印 php.gif
```



```
$img=new Image(); //创建图片对象
$imgname=$img->waterMark("brophp.gif", "php.gif", 5, "wa_"); //加水印
```

## 17.4 文件上传类 FileUpload

文件上传也是开发项目时常见的功能，本类支持单个文件上传，也支持多个文件上传。另外，如果上传的是图片还可以直接进行缩放和加水印的操作。也支持设置文件上传的尺寸和上传文件的类型。如下所示：

```
/*
 * 定义一个控制器类 User
 */
Class User {

    //控制器中添加用户的方法
    function add(){
        $user=D("users");
        $_POST["picname"]=$this->upload();

        $user->insert();
    }

    //文件上传方法
    Private function upload(){
        $up = new FileUpload(); //可以通过参数指定上传位置，可通过 set() 方法

        if($up->upload("pic")) { //pic 为上传表单的名称
            return $up->getFileName(); //返回上传后的文件名
        }else{
            //如果上传失败提示出错原因
            $this->error($up->getErrMsg(), 0, 'index');
        }
    }
}
```

在 FileUpload 类中有几个可以使用的方法：创建对象以后，通过 upload() 方法上传文件，参数为 <input type="file" name="pic"> 的 name 值。如果上传成功可以通过该对象中的 getFileName() 方法获取上传的文件（默认为随机文件名，可以设置）。如果上传失败可以通过 getErrMsg() 方法获取出错信息。还可以通过 set() 方法进行连贯操作，限制上传文件的尺寸、类型和是否启用随机文件名，也可以通过 set() 方法对上传的图片缩放和加水印。如下所示：

```
Private function upload(){
    $up = new FileUpload(); //可以通过参数指定上传位置，可通过 set() 方法

    $up->set("path", "/usr/www/uploads/") //设置上传位置
    ->set("maxSize", 1000000) //设置上传大小，单位字节
    //设置允许上传的类型
    ->set("allowType", array("gif", "jpg", "png"))
    //设置启用上传后随机文件名，true 启用（默认），false 使用原文件名
    ->set("israndname", true)
    //设置上传图片的缩放大小，还可以通过 prefix 指定新名前缀
    ->set("thumb", array("width"=>300, "height"=>200))
```



```
//设置为上传图片加水印，也可以通过 prefix 指定新名前缀
->set("watermark", array("water"=>"php.gif", "position"=>5))

if($sup->upload("pic")) { //pic 为上传表单的名称
    return $sup->getFileName(); //返回上传后的文件名
}else{
    //如果上传失败提示出错原因
    $this->error($sup->getErrMsg(), 0, 'index');
}
}
```

如果是多个文件一起上传 `getFileName()` 返回一个数组，数组是多个上传成功的图片名。如果上传失败 `getErrMsg()` 方法，返回一个多个出错信息数组。

## 18 自定义扩展函数库

使用 BroPHP 框架除了自定义控制器类和业务模型类，还可以自定义一些扩展功能类，只要将类声明在 `classes` 目录下（以 `.class.php` 为后缀名，类名全部小写），并以类名作为文件名，一个文件中存放一个类。如果按这些规范编写，则所有在这个目录下编写的类会被 BroPHP 框架用到时类时自动加载，在任何位置都可以直接创建对象使用（包括静态方法的使用）。

## 19 自定义扩展类库

如果一个很小的功能，就不需要通过编写一个类去完成，只要一个小函数就可以搞定，BroPHP 框架也提供了自定义函数的位置。只要将自定义的功能函数编写在 `commons` 目录下的 `functions.inc.php` 文件中，则在任何位置都可以直接调用。