

目录

简介	3
获取方式	3
官网地址	3
联系作者	4
目录结构	5
部署应用	6
应用入口	7
伪静态	9
运行于 APACHE 的设置方法	9
运行于 NGINX 的设置方法	10
自定义伪静态扩展名	10
自定义伪静态时 URL 参数的分隔符号	11
命名规则	12
命名空间	14
全局函数	16
REST 模式	17
开启 REST 模式	17
REST 模式下 API 接口版本调用方法	18
数据库	19
连接数据库	19
直接调用 MYSQL 类	20
自动连接方法	20
连贯操作增/删/改/查数据	21
缓存	24
MEMCACHED	24
REDIS	24
FILE	25

会话.....	26
布局.....	28
模板引擎.....	29
开启模板引擎.....	29
输出变量.....	29
模板缓存.....	31
上传文件.....	32
日志.....	34
日志存入文件.....	34
日志存入数据库.....	35
附录 A：常量清单.....	36
附录 B：模板标签清单.....	38
{\$变量名}.....	38
{\$数组名.元素}.....	38
{\$对象名.属性}.....	39
{CONST:常量名}.....	39
{LAYOUT:布局名}.....	40
{BEGINBLOCK:代码块名} 和 {ENDBLOCK:代码块名}.....	41
{INSERTBLOCK:代码块名}.....	42

简介

MiniFramework 是一款遵循 Apache2 开源协议发布的，支持 MVC 和 RESTful 的超轻量级 PHP 开发框架。能够帮助开发者用最小的学习成本快速构建 Web 应用，在满足开发者最基础的分层开发、数据库和缓存访问等少量功能基础上，做到尽可能精简，以帮助您的应用基于框架高效运行。

获取方式

GitHub 源码库地址：<https://github.com/jasonweicn/miniframework>

通过 Composer 安装，命令如下：

```
php composer.phar create-project --prefer-dist --stability=dev jasonweicn/miniframework-app-basic
```

提示：上述命令会安装一个基于 MiniFramework 的基础应用模板

官网地址

<http://www.miniframework.com>

联系作者

作者: Jason Wei

信箱: jasonwei06@hotmail.com

博客: <http://www.sunblogger.com>

目录结构

```
|--- App/                                应用案例
|   |--- Api/                            REST 模式的 API
|   |--- Cache/                          缓存
|   |--- Config/                         配置
|       |--- database.php               数据库配置文件（生产环境）
|       |--- database-dev.php           数据库配置文件（开发环境）
|       |--- database-test.php          数据库配置文件（测试环境）
|
|   |--- Controller/                    控制器
|   |--- Layout/                        布局
|   |--- Model/                         模型
|   |--- Public/                        站点根目录
|       |--- css/                       css
|       |--- img/                       图片
|       |--- js/                        js
|       |--- uploads/                   上传文件存储目录
|       |--- index.php                  应用入口文件（生产环境）
|       |--- index-dev.php              应用入口文件（开发环境）
|       |--- index-test.php             应用入口文件（测试环境）
|
|   |--- View/                          视图
|
|--- MiniFramework/                     框架核心目录
    |--- Base/                           基础类库
    |--- Bootstrap.php                   引导程序
```

提示 1: MiniFramework 从 2.0 版开始, 对框架核心进行了重构, 大部分基础类库已移入 Base 目录下。

提示 2: MiniFramework 在 2.7.0 版本中加入了对应用运行环境的支持, 通过常量 APP_ENV 可定义运行环境 (具体请参考示例中的应用入口文件代码)。

部署应用

MiniFramework 支持主程序和 WEB 站点根目录分离部署的特性。

你下载的 MiniFramework 源代码包中，已经附带包含了一个用于演示的应用 demo，目录名为 App（查阅：目录结构），请将 Apache 或 Nginx 的站点根目录指向 App 中的 Public 目录。然后在浏览器中尝试访问如下 URL 地址：

`http://你的域名/index.php?c=index&a=index`

如果，你可以通过浏览器访问类似上面这样的 URL 地址，并获得一个显示有“Hello World!”的页面，这说明你已经部署成功了。

应用入口

使用 MVC 开发模式时，通常需要为应用准备一个入口文件，所有对应用的访问请求都应指向这个入口文件，MiniFramework 也不例外。

在附带的应用案例中，找到 App/Public/index.php，这就是一个入口文件，其代码如下：

```
// 应用命名空间（请与应用所在目录名保持一致）
const APP_NAMESPACE = 'App';

// 是否显示错误信息（默认值：false）
const SHOW_ERROR = true;

// 是否开启日志（生产环境建议关闭，默认值：false）
const LOG_ON = false;

// 是否启用布局功能（默认值：false）
const LAYOUT_ON = true;

// 是否开启 REST 模式的 API 接口功能（默认值：false）
const REST_ON = true;

// 引入 MiniFramework 就是这么简单
require dirname(dirname(__DIR__)) . DIRECTORY_SEPARATOR . 'miniframework/Bootstrap.php';
```

在上边的代码中，最为关键的是最后一行，通过 `require` 命令引入 MiniFramework 的引导程序 `Bootstrap.php`。

在引入引导程序前，你还可以像案例中一样，通过 `const` 命令定义一些

MiniFramework 的关键常量，例如用于显示报错信息的常量 `SHOW_ERROR` 。

提示：MiniFramework 运行所需的全部常量可以在引导程序 *Bootstrap.php* 中找到（1.0.0 版之前引导程序名为 *Mini.php*）。

伪静态

MiniFramework 在设置了 Rewrite 规则后，可实现类似下面这种伪静态访问方式：

http://localhost/Controller/Action/param1/value1/param2/value2

当框架以伪静态模式访问时，URL 中的参数需要通过框架内置方法获取，例如：

```
// 获取全部参数
$params = $this->params->getParams();
```

或者，也可以这样：

```
// 获取指定的参数
$params = $this->params->getParam('param1');
```

提示：伪静态时通过框架内置方法获取 URL 参数的设计初衷是尽量不去污染 `$_GET`。

运行于 Apache 的设置方法

向 Public 目录中添加一个 .htaccess 文件（附带的应用案例中已提供），内容如下：

```
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f
```

```
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^ index.php [L]
```

运行于 Nginx 的设置方法

在 nginx.conf 配置文件中，向 server{} 中添加如下设置：

```
location / {
    index index.html index.php;
    if (!-e $request_filename) {
        rewrite ^/(.*)$ /index.php last;
    }
}
```

提示：MiniFramework 从 1.0.10 版本开始，支持使用下划线作为 URL 分隔符，和支持以 .html 结尾的伪静态 URL，例如：

http://localhost/Controller/Action/param1_value1_param2_value2.html

自定义伪静态扩展名

MiniFramework 从 2.6.0 版本开始，支持通过常量 URL_SUFFIX 定义伪静态扩展名，默认值为"htm|html|shtml"。

例如，可以在应用入口文件 index.php 中进行定义：

```
// 定义伪静态扩展名（多个扩展名用"|"间隔）
const URL_SUFFIX = 'htm|html|shtml';
```

自定义伪静态时 URL 参数的分隔符号

MiniFramework 从 2.6.0 版本开始，支持通过常量 `URL_SPLIT_SYMBOL` 来自定义伪静态时 URL 参数的分隔符号。

例如，可以在应用入口文件 `index.php` 中进行定义：

```
// 定义分隔符号为减号
const URL_SPLIT_SYMBOL = '-';
```

通过上边的定义，便可实现类似下边的伪静态 URL：

`http://localhost/Controller/Action/param1-value1-param2-value2.html`

结合实际应用场景，例如：

`http://localhost/user/info/id-1.html`

命名规则

控制器

只允许使用 a-z、A-Z、0-9 和 _ ， 并以大写字母开头， 例如： Index.php

模型

只允许使用 a-z、A-Z、0-9 和 _ ， 并以大写字母开头， 例如： Info.php

视图

只允许使用 a-z、0-9 和 _ ， 并以字母开头， 例如： index.php

布局

只允许使用 a-z、A-Z、0-9 和 _ ， 并以字母开头， 例如： header.php

API 接口

只允许使用 a-z、A-Z、0-9 和 `_`，并以大写字母开头，例如：Version.php

命名空间

从 1.0.0 版本开始，MiniFramework 已全面启用命名空间，其中：

Mini 对应的框架核心。App 对应你的应用，可以通过在应用的入口文件中，定义常量 APP_NAMESPACE 的值来改变应用的命名空间，例如：

```
const APP_NAMESPACE = 'MyApp'; // 请与应用目录名保持一致
```

创建控制器时，请在页面顶部放置用于声明命名空间的代码，例如：

```
// 声明当前页的命名空间
namespace App\Controller;

// 引入 Action，因为 Action 是框架核心文件，所以前面要加 Mini\
use Mini\Base\Action;

class Index extends Action
{
    function indexAction()
    {
        // do something...
    }
}
```

创建模型时，同样要在页面顶部放置用于声明命名空间的代码，例如：

```
// 声明当前页的命名空间为 App\Model
namespace App\Model;

// 引入框架核心文件
use Mini\Base\Model;
```

```
class Info extends Model
{
    public function getInfo()
    {
        // do something...
    }
}
```

提示: *MiniFramework* 从 2.0 版开始, 对框架核心进行了重构, 大部分基础类库已移入 *Mini\Base* 命名空间下。例如上边的案例中, 引入框架模型类的代码已经变为 *use Mini\Base\Model* 这样的写法了。

全局函数

MiniFramework 在初始化时，会自动加载一个全局函数库，你可以随时调用里面的全局函数，例如：

```
$test = ['a', 'b', 'c'];  
// 调用全局函数 pushJson() 输出一个 JSON 串并终止程序运行  
pushJson($test);
```

提示：全局函数库位于 /Function/Global.func.php

REST 模式

开启 REST 模式

MiniFramework 从 1.0.0 版开始，增加了对 RESTful 的支持，可以在入口文件 `Public/index.php` 中，定义常量 `REST_ON` 的值为 `true` 开启 REST 模式，例如：

```
const REST_ON = true;
```

开启 REST 后，可以访问应用案例中附带的一个名为 `Version` 的 Api 接口 `demo` 进行测试。文件位于 `App/Api/Version.php`，访问方式为：

`http://你的域名/api/version`

访问后，正常情况下会得到如下输出结果：

```
{"code":200,"msg":"success","data":"1.0.0"}
```

需要特别注意的是：如果你的项目中有使用 `Api` 命名的 `Controller`，将会因 REST 开启而失效，所有向 `Api` 的请求均会被指向 `App/Api` 目录。

提示：MiniFramework 的 REST 接口支持输出 JSON 和 XML 两种数据格式，附带的 demo 中已经进行了演示。

REST 模式下 API 接口版本调用方法

MiniFramework 从 1.0.8 版本开始，新增了对于 REST 模式的 API 接口的版本调用方法。可以在发出请求时，向 HEADER 中添加一个名为 Ver 的参数，作用是声明调用的目标接口的版本，其值应为一个整数。MiniFramework 在接到这个请求时，会按 HEADER 中给出的版本号参数 Ver 调用对应的 API 接口文件。

当某个 API 接口需要增加新版本时，开发者需要将对应的接口文件和类名增加后缀 _VX（X 代表版本号），例如：Info_V2.php

提示：最新的 MiniFramework 源代码包中，已经提供了针对 API 接口版本调用的 demo 实例文件，分别位于 /App/Api/Info.php 和 /App/Api/Info_V2.php

数据库

连接数据库

MiniFramework 目前只支持 MySQL 数据库，有手动和自动两种连接方式。

手动连接（工厂模式），代码如下：

```
// 如果未在页面顶部用 use 引入 Db，按照下面的写法，在 Db 前加上 \Mini\Db\  
$db = \Mini\Db\Db::factory ('Mysql',  
    [  
        'host'          => 'localhost', // 主机地址  
        'port'          => 3306,         // 端口  
        'dbname'        => 'mydbname',   // 库名  
        'username'      => 'myuser',     // 用户名  
        'passwd'        => '123456',     // 密码  
        'charset'       => 'utf8',       // 字符编码  
        'persistent'    => false         // 是否启用持久连接 （ true | false ）  
    ]  
);
```

```
// 还可以通过 Config 中的 load() 方法先读取数据库配置，再创建对象  
$dbConfig = \Mini\Base\Config::getInstance()->load('database');  
$db2 = Db::factory ('Mysql', $dbConfig['default']);
```

提示： `Config::getInstance()->load('database')` 这个方法还可以传入 `database:default` 来直接获取 `default` 中的数据（从 1.0.0 版开始支持）

直接调用 MySQL 类

MiniFramework 从 2.0 开始支持直接调用 MySQL 类，这样做的好处是便于让 IDE 对类的方法进行提示，为开发者编码提供便利，代码如下：

```
use Mini\Base\Config;
use Mini\Db\Mysql; // 引入 MySQL 类

$dbParams = Config::getInstance()->load('database:default');
$db = new Mysql($dbParams);
```

自动连接方法

MiniFramework 自动连接数据库功能默认是关闭的，如需使用，请在你的应用入口文件 Public/index.php 中定义常量 DB_AUTO_CONNECT 的值为 true，来开启这个功能，例如：

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

同时，还需要在 Config/database.php 中对数据库连接进行配置，例如：

```
$database['default'] = [
    'host'          => 'localhost', // 主机地址
    'port'          => 3306,         // 端口
    'dbname'        => 'test',       // 库名
    'username'      => 'root',       // 用户名
    'passwd'        => '',           // 密码
    'charset'       => 'utf8',       // 字符编码
    'persistent'    => false        // 是否启用持久连接 ( true | false )
```

```
];
```

接下来，就可以在模型中通过 `$this->loadDb()` 方法直接加载数据库对象了，例如：

```
namespace App\Model;
use Mini\Base\Model;

class Info extends Model // 自动连接数据库，必须继承核心类 Model
{
    public function getInfo()
    {
        // 加载 key 为 default 的数据库
        $db = $this->loadDb('default');

        // do something...
    }
}
```

提示：MiniFramework 的数据库自动连接功能，采用的是惰性连接机制，只会在下达了执行 SQL 语句命令时，才真正开始与数据库通讯建立连接，因此，你不必为开启自动连接功能而担心应用的性能问题。

连贯操作增/删/改/查数据

MiniFramework 从 1.2.0 版本开始，支持在 Model 模型类中，通过“连贯操作”方式增/删/改/查数据，例如：

```
namespace App\Model;
use Mini\Base\Model;

class User extends Model // 继承 Model 模型类
{
    public function getUser()
```

```
{  
    // 设置当前使用的数据库（这里的 default 是数据库连接对象的名称）  
    $this->useDb('default');  
  
    // 示例 1：连贯操作方式向名为 user 的表中插入一条数据纪录  
    $data1 = array('id' => 1, 'name' => '张三');  
    dump($this->table('user')->data($data1)->add());  
  
    // 示例 2：向 user 表中一次插入多条纪录  
    $data2 = array(  
        array('id' => 2, 'name' => '李四'),  
        array('id' => 3, 'name' => '王五')  
    );  
    dump($this->table('user')->data($data2)->add());  
  
    // 示例 3：删除 user 表中 id 为 2 的纪录  
    dump($this->table('user')->where('id=2')->delete());  
  
    // 示例 4：修改 user 表中 id 为 3 的记录  
    dump($this->table('user')->data(array('name' => '赵六'))->where('id=3')->save());  
  
    // 示例 5：查询 user 表中的全部纪录  
    dump($this->table('user')->select());  
  
    // 示例 6：查询 user 表中的全部纪录，但只返回前 2 条纪录  
    dump($this->table('user')->limit(2)->select());  
    // 上方示例 6 中，limit(2) 等价于 limit(0, 2)，用法与 SQL 中的 LIMIT 语法一致  
  
    // 示例 7：查询 user 表中的全部纪录，按 id 字段倒序排列结果  
    dump($this->table('user')->order(array('id' => 'DESC'))->select());  
    // 上方示例 7 中的 order() 方法也可直接传入字符串，例如 order('id DESC')  
  
    // 示例 8：查询 user 表中 id 为 1 的记录  
    $res = $this->table('user')->where('id=1')->select('Row');  
    // 上方示例 8 中，select() 方法传入 Row 参数时返回的结果为键值对形式的一维数组  
  
    // 输出查询结果  
    dump($res);  
  
    // 关于 where 方法，还支持通过参数来构造查询条件，例如下面的示例：  
  
    // 示例 9：查询 id 等于 1 或 2 的记录
```

```
$res = $this->table('user')->where('id', [1, 2])->select();  
// 上边的查询参数, 最终会构造为 id=1 OR id=2, 逻辑运算符默认为“OR”  
  
// 示例 10: 查询 id 不等于 1 且 不等于 2 的记录  
$res = $this->table('user')->where('id', '<>', [1, 2], 'AND')->select();  
// 上边的查询参数, 最终会构造为 id<>1 AND id<>2  
  
// 关于联表查询的示例  
  
// 示例 11: INNER JOIN  
$res = $this->from('user')->innerjoin('user_address',  
'user.id=user_address.id')->select();  
  
// 示例 12: LEFT JOIN  
$res = $this->from('user')->leftjoin('user_address',  
'user.id=user_address.id')->select();  
  
// 示例 13: RIGHT JOIN  
$res = $this->from('user')->rightjoin('user_address',  
'user.id=user_address.id')->select();  
  
// 示例 14: 统一封装的 JOIN 方法, 下边的代码相当于调用 rightjoin 方法  
$res = $this->from('user')->join('user_address', 'user.id=user_address.id',  
'RIGHT')->select();  
// join 方法的第三方参数默认值为"INNER"  
  
}  
}
```

提示 1: MiniFramework 在 2.1.0 开始支持 *add*、*save*、*delete* 和 *data* 四个连贯操作的方法, 并对原有的 *order*、*limit*、*group* 和 *select* 方法进行了改进和完善。

提示 2: MiniFramework 2.5.0 中针对 *where* 方法新增了通过参数构造查询条件的特性。

提示 3: MiniFramework 2.7.0 中, 新增了更符合习惯的 *from* 方法, 与原 *table* 方法的用法完全一致。

缓存

MiniFramework 支持三种缓存方式，分别是：Memcached、Redis 和 File（磁盘文件存储）。

Memcached

```
// 如 PHP 安装的是 Memcached 扩展，就传入 'Memcached'
$cache = \Mini\Cache\Cache::factory ('Memcache',
    array (
        'host'      => 'localhost', //主机
        'port'      => 11211         //端口
    )
);

// 写入一个名为 test 的缓存，值为 abc，有效时间为 3600 秒
$cache->set('test', 'abc', 3600);

// 读取名为 test 的缓存
$test = $cache->get('test');
```

提示：可以通过 `getMemcacheObj()` 来获得 Memcache 对象，以调用未在框架中封装的 Memcache 更多的方法。

Redis

```
$cache = \Mini\Cache\Cache::factory ('Redis',
    array (
        'host'      => 'localhost', //主机
        'port'      => 11211,       //端口
    )
);
```



```
'passwd' => '' //密码
)
);

// 写入一个名为 test 的缓存，值为 abc，有效时间为 3600 秒
$cache->set('test', 'abc', 3600);

// 读取名为 test 的缓存
$test = $cache->get('test');
```

提示：可以通过 `getRedisObj()` 来获得 `Redis` 对象，以调用未在框架中封装的 `Redis` 更多的方法。

File

```
$cache = \Mini\Cache\Cache::factory('File');

// 写入一个名为 test 的缓存，值为 abc，有效时间为 3600 秒
$cache->set('test', 'abc', 3600);

// 读取名为 test 的缓存
$test = $cache->get('test');
```

提示：使用 `File` 类型的缓存时，缓存文件会存储在默认路径 `App/Cache/` 中，你可以在应用入口文件 `App/Public/index.php` 中，定义常量 `CACHE_PATH` 的值来改变存储路径。

会话

MiniFramework 从 1.0.12 版本开始，新增了 Session 会话类。

示例代码如下：

```
namespace App\Controller;
use Mini\Base\Session;

class Example extends Action
{
    function sessionAction()
    {
        // 开启会话
        Session::start();

        // 写入一个名为 test 的会话，对应的值为 abc
        Session::set('test', 'abc');

        // 读取名为 test 的会话
        $test = Session::get('test');

        dump($test);
        die();
    }
}
```

Session 会话类还支持在开启时传入针对 SESSION 的设定参数，例如：

```
// 开启会话
\Mini\Base\Session::start(array(

    // 设定 SESSION 存储于 Memcached
    'save_handler' => 'memcache',
```

```
// Memcached 主机地址和端口
'save_path'    => '127.0.0.1:11211'

));
```

布局

MiniFramework 的布局 (Layout) 功能为开发者提供了让多个 View 具有相同部分的机制，例如具有相同的 header 或 footer 代码。

布局功能默认是关闭的，如需使用，请在你的应用入口文件 App/Public/index.php 中定义常量 LAYOUT_ON 的值为 true 开启布局功能，例如：

```
const LAYOUT_ON = true;
```

提示：附带的应用案例中，已经开启了布局功能，并演示了如何使用布局。

模板引擎

MiniFramework 2.8.0 版本提供了模板引擎功能，开发者可以通过特定的标记符号，在 View 和 Layout 中预定义变量输出等模板标记。

开启模板引擎

开发者可以通过在应用的入口文件中定义常量 TPL_ON 的值为 true（默认值为：false），即可开启模板引擎，例如：

```
const TPL_ON = true;
```

提示：附带的应用案例中，已经开启了应用引擎。

作者建议：开发 MiniFramework 的核心思想是提供一个精简的微型框架，模板引擎实际上是与精简思想背道而驰的，通过“发明”一套新的标记语法来替代原本可以通过 PHP 直接实现的功能，拖累了框架的运行效率。因此，模板引擎是默认关闭的，希望开发者可以优先考虑 PHP 原生写法来在 View 和 Layout 中实现所需功能，原生 PHP 已经很优秀了。

输出变量

开发者可以在 View 和 Layout 中通过特定的标记符号对变量进行标记，模板引擎通过对开发者编写的标记符号，编译后，实现在 View 和 Layout 中按指定的位置输出变量的值，例如：

在 Controller 中向 View 传递变量

```
// 向 View 传递一个名为 title 的变量
$this->view->assign('title', '这里是标题文本');

// 渲染并显示 View
$this->view->display();
```

在 View 中标记变量

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>{$title}</title>
<link rel="stylesheet" type="text/css" href="{ $baseUrl } /css/default.css">
</head>
```

提示：上述代码中，`{ $baseUrl }`是一个框架默认的变量，用于输出基础路径。（更多模板标记可参考附带的[应用案例](#)）

经过模板引擎编译后为：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title><?php echo $this->title; ?></title>
<link rel="stylesheet" type="text/css" href="<?php echo $this->baseUrl; ?>/css/default.css">
</head>
```

模板缓存

在开启模板引擎的状态下，MiniFramework 会将 View 和 Layout 的文件进行解析编译，并将编译后的代码以缓存文件的形式，保存在常量 `CACHE_PATH` 定义的缓存目录中。

开发者修改 View 和 Layout 文件后，模板引擎会比对文件最后修改时间，并对缓存文件进行更新。

上传文件

MiniFramework 从 1.2.0 版本开始, 新增了支持通过自带的 Upload 类实现文件的上传保存。示例代码如下:

```
// 实例化 Upload 类
$upload = new \Mini\Base\Upload();

// 调用 save 方法, 将要保存的文件作为参数传入
$res = $upload->save($_FILES['file']);

// 保存成功时 $res 中的返回值为保存路径和文件名, 失败时值为 false
if (!$res) {
    $errmsg = $upload->getErrMsg();
    echo $errmsg;
} else {
    dump($res);
}
```

提示: 保存失败时, 可以通过 `getErrMsg()` 方法获取错误信息。

在实例化 Upload 类时, 可传入一个数组类型的参数, 对文件保存路径、大小和类型进行设定, 例如:

```
// 配置数组
$config = array(

    // 文件保存的根目录
    'rootPath' => PUBLIC_PATH . DS . 'uploads',

    // 文件的大小限制 (单位: Byte)
    'maxPath' => 512000,
```



```
// 允许的类型
'allowType' => 'bmp, gif, jpg, jpeg, png'

);

// 实例化 Upload 类时，将配置数组作为参数传入
$upload = new \Mini\Base\Upload($config);
```

提示：上面示例代码中，配置项可有选择的进行设定，没有设定的，框架会使用默认值处理。

日志

MiniFramework 从 1.4.0 版本开始，新增了日志功能，开发者可通过在应用的入口文件中定义 LOG_ON 为 true 来开启日志功能。示例代码如下：

```
// 开启日志（生产环境建议关闭）  
const LOG_ON = true;
```

提示：如果在入口文件中未对常量 LOG_ON 进行声明，则其默认值为 false。

日志存入文件

开发者可通过定义 LOG_MODE 和 LOG_PATH 两个常量来实现将日志存入文件，例如：

```
// 首先要激活日志功能（默认值为：false）  
const LOG_ON = true;  
  
// 定义日志的存储模式（默认值为：1，1 为文件，2 为数据库）  
const LOG_MODE = 1;  
  
// 定义日志的存储路径  
const LOG_PATH = '/data/htdocs/myapp/logs';
```

通过上边的代码，便实现了将日志以文件的形式存储到指定的路径下。当开发者编写的代码运行遇到错误时，MiniFramework 会将捕获到的错误信息按 yyyy-mm-dd.log 的文件命名方式进行存储，例如：

/data/htdocs/myapp/logs/2021-01-06.log

日志存入数据库

MiniFramework 从 2.6.0 版本开始支持日志存储到数据库的特性。开发者可通过定义 LOG_MODE、LOG_DB_CONFIG 和 LOG_TABLE_NAME 三个常量来实现将日志存入文件，例如：

```
// 首先要激活日志功能（默认值为：false）
const LOG_ON = true;

// 定义日志的存储模式（默认值为：1，1 为文件，2 为数据库）
const LOG_MODE = 2;

// 定义日志存储所使用的数据库配置
const LOG_DB_CONFIG = 'database:default';

// 定义日志存储的数据表名
const LOG_TABLE_NAME = 'myapp_log';
```

附录 A：常量清单

(按常量名称字母顺序排列)

常量名	用途	支持版本	备注
APP_ENV	定义应用运行环境	V2.7.0 及以上	默认值: "prod"
APP_NAMESPACE	定义应用的命名空间名称	V1.0.0 及以上	默认值: "App"
APP_PATH	定义应用路径	V0.1.0 及以上	
CACHE_PATH	定义缓存文件存储路径	V0.7.0 及以上	
CONFIG_PATH	定义配置文件读取路径	V0.10.0 及以上	
CSRF_TOKEN_ON	CSRF 令牌功能开关	V2.0.0 及以上	默认值: TRUE, 在 2.0 版新增, 从 2.4.0 版开始默认值变更为: FALSE
CSRF_TYPE	定义客户端获取 CSRF 令牌的方式	V2.4.0 及以上	默认值: "cookie"
DB_AUTO_CONNECT	数据库自动连接功能开关	V0.10.0 及以上	默认值: FALSE
DS	定义系统目录分隔符	V1.0.0 及以上	
HTTP_CACHE_CONTROL	定义 HTTP 缓存指令	V0.10.0 及以上	默认值: "private"
LAYOUT_ON	布局功能开关	V0.9.0 及以上	默认值: FALSE
LAYOUT_PATH	定义布局文件读取路径	V0.9.0 及以上	
LIB_PATH	定义框架核心类库读取路径	V1.0.0~V1.5.2	在 2.x 中已删除
LOG_DB_CONFIG	定义日志存储的数据库配置	V2.6.0 及以上	默认值: "database:default"
LOG_LEVEL	定义日志记录等级	V1.4.0 及以上	
LOG_MODE	定义日志存储模式, 1 为文件, 2 为数据库	V2.6.0 及以上	默认值: 1
LOG_ON	日志功能开关	V1.4.0 及以上	默认值: FALSE

LOG_PATH	定义日志存储路径	V1.4.0 及以上	
LOG_TABLE_NAME	定义日志存储的数据表名	V2.6.0 及以上	默认值: "log"
MINI_PATH	定义框架核心文件读取路径	V2.0.0 及以上	
PUBLIC_PATH	定义 WEB 站点目录对应的磁盘路径	V1.2.0 及以上	
REST_ON	REST 功能开关	V1.0.0 及以上	默认值: FALSE
SHOW_DEBUG	是否显示开发者调试信息开关	V1.0.0 及以上	默认值: TRUE
SHOW_ERROR	是否显示错误信息开关	V0.3.0 及以上	默认值: FALSE
TPL_ON	模板引擎开关	V2.8.0 及以上	默认值: FALSE
TPL_SEPARATOR_L	模板引擎标记开始符号	V2.8.0 及以上	默认值: "{"
TPL_SEPARATOR_R	模板引擎标记结束符号	V2.8.0 及以上	默认值: "}"
URL_SPLIT_SYMBOL	定义伪静态 URL 参数分割符号	V2.6.0 及以上	默认值: "_"
URL_SUFFIX	定义伪静态扩展名	V2.6.0 及以上	

附录 B：模板标签清单

`{ $变量名 }`

用途说明：标记输出一个通过 `$this->view->assign()` 方法传入 View 的变量。

示例代码：

```
<p>{ $info }</p>
```

编译输出：

```
<p><?php echo $this->info; ?></p>
```

`{ $数组名.元素 }`

用途说明：标记输出一个通过 `$this->view->assign()` 方法传入 View 的数组元素。

示例代码：

```
<p>姓名： { $user.name }</p>  
<p>性别： { $user.sex }</p>
```

编译输出：

```
<p>姓名: <?php echo $this->user["name"]; ?></p>
<p>性别: <?php echo $this->user["sex"]; ?></p>
```

{**\$对象名.属性**}

用途说明: 标记输出一个通过 `$this->view->assign()` 方法传入 View 的对象属性。

示例代码:

```
<p>姓名: {$user.name}</p>
<p>性别: {$user.sex}</p>
```

编译输出:

```
<p>姓名: <?php echo $this->user->name; ?></p>
<p>性别: <?php echo $this->user->sex; ?></p>
```

{**const:常量名**}

用途说明: 标记输出一个常量。

示例代码:

```
<p>这是一个常量: {const:APP_NAME}</p>
```

编译输出：

```
<p>这是一个常量：MiniFramework</p>
```

提示：如果常量存在，会直接将值输出

{layout:布局名}

用途说明：标记加载布局

示例代码：

```
{layout:header}  
<body>  
    {layout:content}  
</body>  
{layout:footer}
```

编译输出：

```
<?php echo $this->_layout->header; ?>  
<body>  
    <?php echo $this->_layout->content; ?>  
</body>  
<?php echo $this->_layout->footer; ?>
```

提示：模板引擎会检查并渲染 Layout

{beginBlock:代码块名} 和 {endBlock:代码块名}

用途说明：标记一个代码块区域

示例代码：

```
{beginBlock:myblock}
<script>
/*
这是一个 Block 的示例
    通常我们会希望 js 代码放到页面底部运行，
    在使用布局的情况下，可以在 View 中通过 beginBlock 和 endBlock 预定义一个代码块，
    在 Layout 文件中，可以通过 insertBlock 将对应的代码块插入到需要的地方。（请见
Layout/default.php）
*/
console.log('this is block');
</script>
{endBlock:myblock}
```

编译输出：

```
<?php $this->beginBlock("myblock"); ?>
<script>
/*
这是一个 Block 的示例
    通常我们会希望 js 代码放到页面底部运行，
    在使用布局的情况下，可以在 View 中通过 beginBlock 和 endBlock 预定义一个代码块，
    在 Layout 文件中，可以通过 insertBlock 将对应的代码块插入到需要的地方。
*/
console.log('this is block');
</script>
<?php $this->endBlock("myblock"); ?>
```

提示：结束代码块可以省略名称简写为 {\$endBlock}

{insertBlock:代码块名}

用途说明： 标记插入指定名称的代码块区域的代码。

示例代码：

```
<html>
  <body>
    <p>如果这里是 Layout，可以将在 View 中定义的代码块插入到当前的 Layout 中。</p>
  </body>
</html>
{insertBlock:myblock}
```

编译输出：

```
<html>
  <body>
    <p>如果这里是 Layout，可以将在 View 中定义的代码块插入到当前的 Layout 中。</p>
  </body>
</html>
<?php $this->insertBlock("myblock"); ?>
```