模块五：MVC 开发模式

MVC 是 Xerox PRAC（施乐帕克研究中心）在 20 世纪 80 年代为编程语言 Smalltalk – 80 发明的一种软件设计模式，至今已被广泛使用。随着互联网的发展，对于 Web 应用的功能需求越来越复杂，MVC 在 Web 开发领域也备受欢迎。接下来，本模块将针对 MVC 开发模式进行详细地讲解。

通过本模块的学习，读者对于知识的掌握程度要达到如下目标：

- ✓ 理解 MVC 的概念，可以描述 MVC 思想和工作流程
- ✓ 掌握模型、视图、控制器的创建，理解自动加载与请求分发机制
- ✓ 掌握 MVC 框架的典型实现，能够运用 MVC 框架进行项目开发

任务一：认识 MVC

MVC 是目前广泛流行的一种软件开发模式。利用 MVC 可以将程序中的功能实现、数据处理和界面显示相分离，从而在开发复杂的应用程序时，开发者可以专注于其中的某个方面，进而提高开发效率和项目质量。

MVC 这个名字是由 Model、View、Controller 这三个单词的首字母组成的，它表示将软件系统分成三个核心部件：模型（Model）、视图（View）、控制器（Controller），分别用于处理各自的任务。

在用 MVC 进行的 Web 应用开发中，模型是指处理数据的部分，视图是指显示在浏览器中的网页，控制器是指处理用户交互的程序。例如，提交表单时，由控制器负责读取用户提交的数据，然后向模型发送数据，再通过视图将处理结果显示给用户。MVC 的工作流程如图 3-22 所示。

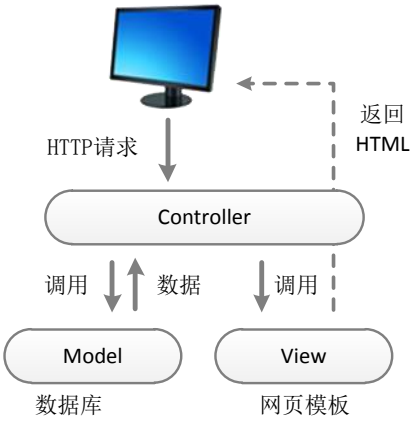

图3-22　MVC 的工作流程

从图 3-22 中可以看出，浏览器向服务器端的控制器发送了 HTTP 请求，控制器就会调用模型来取得数据，然后调用视图，将数据分配到网页模板中，再将最终结果的 HTML 网页返回给浏览器。

MVC 是优秀的设计思想，使开发团队能够更好地分工协作，显著提高工作效率。但是对于小型项目，如果严格遵循 MVC，会增加结构的复杂性，增加工作量，降低运行的效率，因此 MVC 不适用于小型项目。MVC 提倡模型和视图分离，这样也会给调试程序带来一定的困难，每个构件在使用之前都需要经过彻底的测试。尽管 MVC 有一些缺点，但其带来的好处远远超过了这些缺点。对于大型 Web 应用程序，MVC 开发模式可以发挥出巨大的优势。

任务二：MVC 典型实现

MVC 是应用在实际项目中的开发模式。为了更好地学习这种模式，本节将结合博学谷云课堂项目的实际开发，讲解 MVC 项目的典型实现。

1、 数据库操作类

在面对复杂问题时，面向对象编程可以更好地描述现实中的业务逻辑，所以 MVC 应用也是通过面向对象方式实现的。接下来，为项目创建一个数据库操作类 MySQLPDO.class.php，具体代码如下。

```
<?php
//基于 PDO 扩展的 MySQL 数据库操作类
class MySQLPDO {
    protected static $db = null;        //保存 PDO 实例
    public function __construct(){
        self::$db || self::_connect(); //实例化 PDO 对象
    }
    private function __clone() {}       //阻止克隆
    //连接目标服务器（只在构造方法中调用一次）
    private static function _connect(){
        $config = $GLOBALS['dbConfig']; //通过全局变量获取数据库配置信息
        //准备 PDO 的 DSN 连接信息
        $dsn = "{$config['db']}:host={$config['host']};port={$config['port']};
        dbname={$config['dbname']};charset={$config['charset']}";
        try{  //连接数据库
            self::$db = new PDO($dsn, $config['user'], $config['pass']);
        }catch (PDOException $e){
            exit('数据库连接失败：'.$e->getMessage());
        }
    }
    /**
     * 通过预处理方式执行 SQL
     * @param string $sql 执行的 SQL 语句模板
     * @param array $data 数据部分
     * @return object PDOStatement
     */
    public function query($sql, $data=[]){
        //通过预处理方式执行 SQL
        $stmt = self::$db->prepare($sql);
        //批量执行
        is_array(current($data)) || $data = [$data];  //自动转换为二维数组
        foreach($data as $v){
            if(false === $stmt->execute($v)){
                exit('数据库操作失败：'.implode('-', $stmt->errorInfo()));
            }
        }
```

```
37          return $stmt;
38      }
39      //执行 SQL，返回受影响的行数
40      public function exec($sql, $data=[]){
41          return $this->query($sql, $data)->rowCount();
42      }
43      //取得所有结果
44      public function fetchAll($sql, $data=[]){
45          return $this->query($sql, $data)->fetchAll(PDO::FETCH_ASSOC);
46      }
47      //取得一行结果
48      public function fetchRow($sql, $data=[]){
49          return $this->query($sql, $data)->fetch(PDO::FETCH_ASSOC);
50      }
51      //取得一列结果
52      public function fetchColumn($sql, $data=[]){
53          return $this->query($sql, $data)->fetchColumn();
54      }
55      //最后插入的 ID
56      public function lastInsertId(){
57          return self::$db->lastInsertId();
58      }
59  }
```

上述代码是一个基于 PDO 扩展的 MySQL 数据库操作类，类中封装了 PHP 访问 MySQL 数据库的一些基本操作。第 27~38 行代码实现了以 PDO 预处理的方式执行 SQL 语句，并且支持批量操作。

为了测试数据库操作类是否正确执行，接下来创建 index.php 进行测试，具体代码如下。

```
1  <?php
2  //载入数据库操作类
3  require './MySQLPDO.class.php';
4  //准备数据库连接信息
5  $dbConfig = [
6      'db' => 'mysql',            //数据库类型
7      'host' => 'localhost',      //服务器地址
8      'port' => '3306',           //端口
9      'user' => 'root',           //用户名
10     'pass' => '123456',         //密码
11     'charset' => 'utf8',        //字符集
12     'dbname' => 'itcast_bxg'    //默认数据库
13 ];
14 //实例化数据库操作类
15 $db = new MySQLPDO();
16 //执行 SQL 语句并显示执行结果
17 echo '<pre>';
18 var_dump($db->fetchAll('SHOW TABLES'));
```

```
19 echo '</pre>';
```

在浏览器中访问 index.php，运行结果如图 3-23 所示。从图中可以看出，数据库操作类成功执行 SQL 语句，返回了关联数组形式的查询结果。

图3-23　数据库查询结果

2、 模型

模型是处理数据的，而数据是存储在数据库中的。在项目中，所有对数据库的操作都是由模型类来完成的。MVC 中的模型，其实就是为项目中的每个表建立一个模型。

接下来以项目中的栏目表为例，创建栏目表模型类文件 CategoryModel.class.php，具体代码如下：

```
1  <?php
2  //栏目表的模型类
3  class CategoryModel extends MySQLPDO {
4      //获取所有的栏目数据
5      public function getData(){
6          return $this->fetchAll('SELECT * FROM bxg_category');
7      }
8      //添加一个栏目，返回添加后的 ID
9      public function addData($name, $pid){
10         $data = ['name'=>$name, 'pid'=>$pid];
11         $this->query('INSERT INTO bxg_category (name,pid) VALUES(:name,:pid)', $data);
12         return $this->lastInsertId();
13     }
14 }
```

上述代码创建了栏目模型，并实现了栏目查询与栏目添加两个方法。由于模型类继承了数据库操作类，因此模型类可以直接调用数据库操作类中的方法执行 SQL 语句。

接下来编写 index.php 测试模型类是否正确执行，具体代码如下。

```
1  <?php
2  //载入数据库操作类、模型类
3  require './MySQLPDO.class.php';
4  require './CategoryModel.class.php';
5  //准备数据库连接信息
6  //……
```

```
7  //实例化模型类
8  $Category = new CategoryModel();
9  //添加一个栏目，显示添加后的结果
10 $Category->addData('phone','0');
11 echo '<pre>';
12 var_dump($Category->getData());
13 echo '</pre>';
```

在浏览器中访问 index.php，运行结果如图 3-24 所示。从图中可以看出，模型类成功完成了数据库操作。

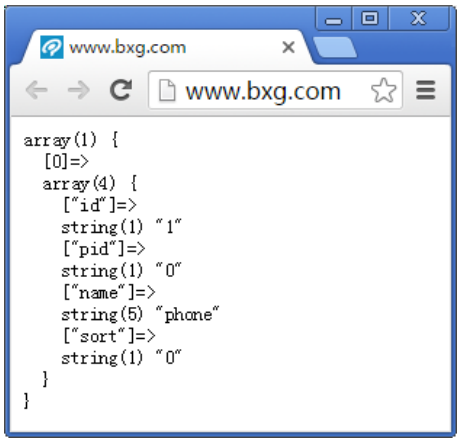

图3-24　测试模型类

3、 控制器

控制器是 MVC 应用程序中的指挥官，它接收用户的请求，并决定需要调用哪些模型进行处理，再用相应的视图显示从模型返回的数据，最后通过浏览器呈现给用户。

如果用面向对象的方式实现控制器，就需要先理解模块的概念。一个成熟的项目是由多个模块组成的，每个模块又是一系列相关功能的集合。以栏目管理模块为例，功能划分如图 3-25 所示。

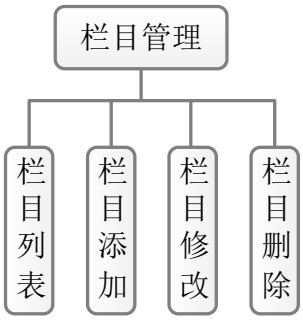

图3-25　栏目管理模块

正如模型是根据数据表创建的一样，控制器则是根据模块创建的，即每个模块对应一个控制器类，模块中的功能都在控制器类中完成。因此，控制器类中定义的方法，就是模块中的功能。

接下来为栏目模块创建控制器类 CategoryController.class.php，具体代码如下：

```
1  <?php
2  //栏目控制器
3  class CategoryController {
4      //栏目列表
5      public function indexAction(){
6          $Category = new CategoryModel();    //实例化栏目模型
```

```
7           $data = $Category->getData();       //查询栏目数据
8           require './index.html';             //载入视图
9       }
10      //栏目添加
11      public function addAction(){}
12      //栏目修改
13      public function editAction(){}
14      //栏目删除
15      public function delAction(){}
16  }
```

上述代码在栏目控制器中定义了栏目列表、栏目添加、栏目修改和栏目删除 4 种方法，命名时使用“Action”后缀。其中栏目列表从数据库中获取了栏目数据，然后载入视图模板文件进行页面显示。

4、 视图

视图是 MVC 中用于显示的网页。通常开发者编写的视图是一个 HTML 模板，在模板中输出来自数据库中的数据。接下来编写栏目列表的视图模板 index.html，其关键代码如下。

```
1  <table>
2      <tr><th>ID</th><th>栏目名</th></tr>
3      <?php foreach($data as $v): ?>
4          <tr><td><?=$v['id']?></td><td><?=$v['name']?></td></tr>
5      <?php endforeach; ?>
6  </table>
```

上述代码实现了将数据库查询出的栏目列表数据输出到网页的<table>表格中。此外，由于栏目控制器使用“require”加载了此视图文件，因此在视图中可以直接使用控制器中的变量$data。

5、 前端控制器

前端控制器是指项目的入口文件 index.php。使用 MVC 模式开发的是一种单一入口的应用程序。传统的 Web 程序是多入口的，即通过访问不同的 PHP 文件来完成用户请求。例如，管理栏目时访问 category.php，管理课程时访问 course.php。单入口程序指的是只有一个 index.php 提供用户访问。

前端控制器又称请求分发器（dispather），通过 URL 参数判断用户请求了哪个功能，然后完成相关控制器的加载、实例化、方法调用等操作。接下来通过一个图例来演示请求分发的流程，如图 3-26 所示。

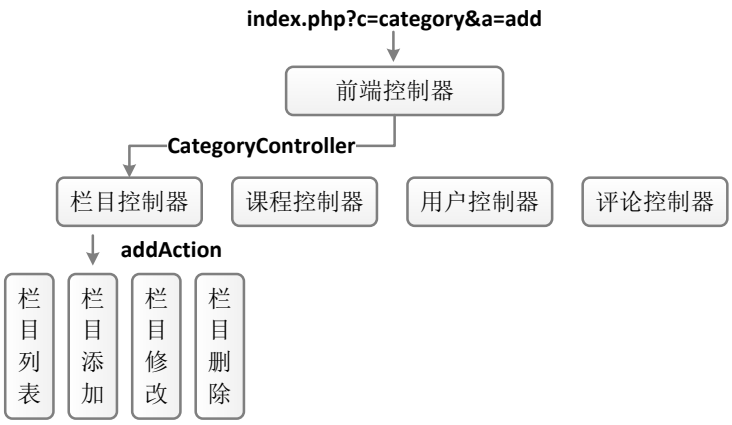

图3-26 请求分发的流程

在图 3-26 中，前端控制器 index.php 接收到两个 GET 参数：c 和 a，c 代表 Controller（控制器），a 代表 Action（操作），所以“c=category&a=add”表示 Category 控制器中的 addAction 方法。

接下来编写 index.php 实现前端控制器，具体代码如下：

```
<?php
//载入数据库操作类、模型类，准备数据库连接信息
//……
//获取控制器、操作名称
$c = isset($_GET['c']) ? ucwords($_GET['c']) : '';
$a = isset($_GET['a']) ? $_GET['a'] : '';
//为名称添加后缀
$c_name = $c.'Controller';
$a_name = $a.'Action';
//请求分发
require "./{$c_name}.class.php"; //载入控制器文件
$Controller = new $c_name();      //实例化控制器
$Controller->$a_name();           //调用方法
```

上述代码通过 GET 参数实现了前端控制器的请求分发。为了测试程序是否正确运行，下面通过浏览器访问“index.php?c=category&a=index”，运行结果如图 3-27 所示。从图中可以看出，浏览器访问到 Category 控制器中的 indexAction 方法，并在视图模板中成功的输出了查询结果。

图3-27　前端控制器运行结果

至此，MVC 开发模式的典型实现已经完成。通过 MVC 开发模式，实现了模型、视图、控制器三者的分离，增强了代码的可维护性，有利于团队开发时的分工协作。

任务三：MVC 框架

框架在软件系统中是一个代码骨架，其作用是通过设计一个所有项目通用的底层代码，来提高项目的开发效率。以盖房子来说，框架相当于已经盖好的房子，但是内部没有装修，当需要开一个水果店时，可以把这个房子装修成水果店，而不需要重新盖一个房子。

通过 MVC 开发模式，可以将整个项目分成应用（application）与框架（framework）两部分，在应用中处理与当前站点相关的业务逻辑，在框架中封装所有项目的底层代码。本节将针对 MVC 框架进行详细讲解。

1、 项目结构

前面创建的模型、控制器、视图文件都保存到一个目录中，在实际项目中显然不能这样做，需要一个合理的目录结构来管理这些文件。接下来演示一种常见的 MVC 目录划分方式，如图 3-28 所示。

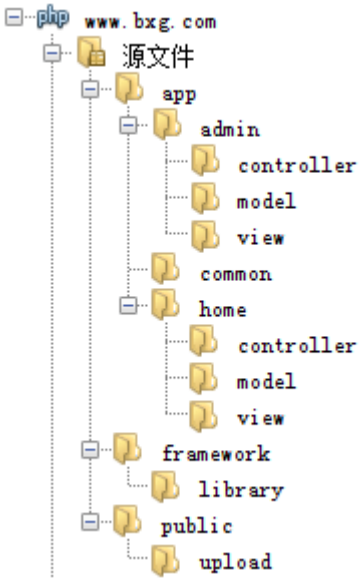

图3-28　MVC 的目录划分

在图 3-28 中，项目主要划分成 app 和 framework 两个目录，app 表示应用（application），用于存放与当前站点业务逻辑相关的文件，framework 表示框架，存放项目的底层文件。app 下的 admin 和 home 目录代表网站的平台，其中 admin 表示后台，为管理员提供管理功能，home 表示前台，为用户提供服务。前台和后台下都有 controller、model 和 view 目录，用于存放与之相关的代码文件。

接下来，将前面创建的数据库操作类、模型类、控制器类、视图文件、入口文件以图 3-28 所示的目录结构进行分配，分配后的结果如表 3-15 所示。

表3-15　MVC 框架项目结构

文件路径	文件描述
index.php	入口文件
app\common	应用公共文件目录
app\home\controller	前台控制器目录
app\home\model	前台模型目录
app\home\view	前台视图目录
app\home\controller\CategoryController.class.php	前台栏目控制器类
app\home\model\CategoryModel.class.php	前台模型类
app\home\view\category\index.html	前台栏目控制器下的视图文件
app\admin\controller	后台控制器目录
app\admin\model	后台模型目录
app\admin\view	后台视图目录
framework\library	框架类库目录
framework\library\MySQLPDO.class.php	数据库操作类
public	公开文件目录（保存 css、images、js 文件）
public\upload	上传文件保存目录

2、 框架基础类

在项目的初始化阶段，需要完成设置常量、载入类库、请求分发等操作。这些都是项目中的底层代码，可以封装一个框架基础类来完成这些任务。下面通过一个图例来演示框架基础类的工作流程，如图 3-29 所示。从图中可看出，框架基础类封装了设置常量、载入类库和请求分发的工作，而入口文件只需要调用框架基础类即可完成任务。

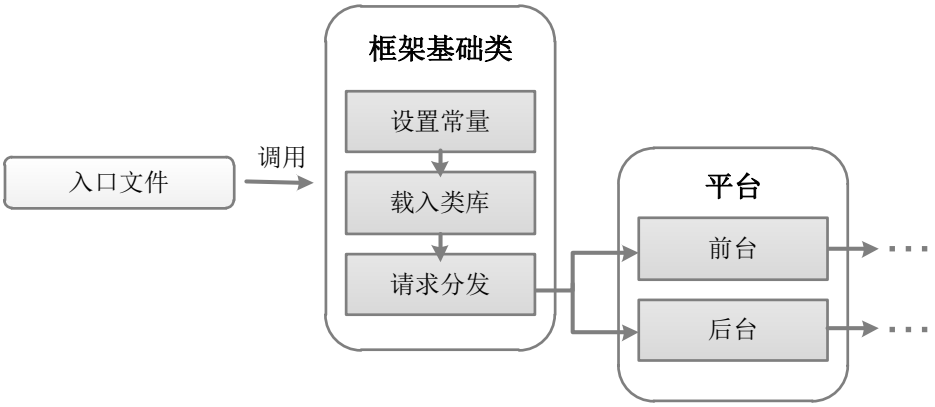

图3-29 框架基础类工作流程

接下来开始编写框架基础类，在 framework 目录中创建文件 Framework.class.php，编写代码如下。

```
<?php
//框架基础类
class Framework{
    //启动项目
    public static function run(){
        self::_init();              //初始化
        self::_registerAutoLoad(); //注册自动加载
        self::_extend();            //扩展功能
        self::_dispatch();          //请求分发
    }
    //初始化
    private static function _init(){
        //设置常量供项目内使用
        define('DS', DIRECTORY_SEPARATOR);                      //路径分隔符
        define('ROOT', getcwd().DS);                            //项目根目录
        define('APP_PATH', ROOT.'app'.DS);                      //应用目录
        define('FRAMEWORK_PATH', ROOT.'framework'.DS);          //框架目录
        define('LIBRARY_PATH', FRAMEWORK_PATH.'library'.DS);   //类库目录
        define('COMMON_PATH', APP_PATH.'common'.DS);            //公共目录
        //获取 p、c、a 参数
        list($p,$c,$a) = self::_getParams();
        define('PLATFORM', strtolower($p));
        define('CONTROLLER', strtolower($c));
        define('ACTION', strtolower($a));
        //拼接平台、控制器、模型、视图路径
        define('PLATFORM_PATH', APP_PATH.PLATFORM.DS);              //平台目录
        define('CONTROLLER_PATH', PLATFORM_PATH.'controller'.DS); //控制器目录
        define('MODEL_PATH', PLATFORM_PATH.'model'.DS);             //模型目录
        define('VIEW_PATH', PLATFORM_PATH.'view'.DS);               //视图目录
        //视图路径
        define('COMMON_VIEW', VIEW_PATH.'common'.DS);
        define('CONTROLLER_VIEW', VIEW_PATH.CONTROLLER.DS);
```

```
33          define('ACTION_VIEW', CONTROLLER_VIEW.ACTION.'.html');
34      }
35      //注册自动加载
36      private static function _registerAutoLoad(){
37          spl_autoload_register(function($class_name){
38              $class_name = ucwords($class_name);
39              if(strpos($class_name, 'Controller')){
40                  $target = CONTROLLER_PATH."$class_name.class.php";
41              }elseif(strpos($class_name, 'Model')){
42                  $target = MODEL_PATH."$class_name.class.php";
43              }else{
44                  $target = LIBRARY_PATH."$class_name.class.php";
45              }
46              require $target;
47          });
48      }
49      //扩展功能（该方法以后实现）
50      private static function _extend(){}
51      //请求分发
52      private static function _dispatch(){
53          $c = CONTROLLER.'Controller';
54          $a = ACTION.'Action';
55          //实现请求分发
56          $Controller = new $c();     //实例化控制器
57          $Controller->$a();          //调用操作
58      }
59      //获取请求参数
60      private static function _getParams(){
61          //获取 URL 参数
62          $p = isset($_POST['p']) ? $_POST['p'] : 'home');
63          $c = isset($_POST['p']) ? $_POST['p'] : 'index');
64          $a = isset($_POST['p']) ? $_POST['p'] : 'index');
65          return [$p, $c, $a];
66      }
67  }
```

上述代码在类中封装了设置常量、载入类库、请求分发三大功能，并提供了一个 run()方法执行调用。自动加载使用了 spl_autoload_register()函数，该函数可以传递一个回调函数作为参数。请求分发实现了从 GET 参数中获取平台、控制器、方法三个请求参数，并支持默认参数。在加载类文件时，程序会先根据类名后缀在 app 目录中进行加载，如果没有匹配的后缀，则加载框架类库目录中的类文件。

在框架基础类完成常量设置后，当需要在控制器中载入视图时，可以直接通过常量 ACTION_VIEW 找到相应的视图文件。修改文件 app\controller\CategoryController.class.php，代码如下。

```
1  <?php
2  class CategoryController {
3      public function indexAction(){
```

```
4           //……
5           require ACTION_VIEW;        //载入视图
6       }
7   }
```

在上述代码中，第 7 行将载入视图的代码直接写为“require ACTION_VIEW”，即可自动载入位于 app\home\view\category\index.html 的视图文件。

在实现框架基础类后，下面在项目入口文件 index.php 中调用框架基础类，代码如下。

```
1  <?php
2  define('APP_DEBUG', true);                      //项目调试开关
3  require './framework/Framework.class.php';      //载入框架基础类
4  Framework::run();                               //运行项目
```

经过上述修改后，MVC 框架已经搭建完成。项目中的 framework 目录可以用于开发任何一个 PHP 项目，由此增强了代码的可复用性。

3、 函数与配置文件

在项目开发时，有许多常用的功能可以通过函数来完成，因此可以在 MVC 框架中编写一个函数库，用于保存项目中的常用函数。在前面开发的“项目二”中，已经编写了项目常用函数库 function.php，将该文件直接复制到本项目的 framework 目录中即可。

将函数库文件复制完成后，接下来修改 framework\Framework.class.php 文件，在初始化的方法中载入函数库，具体代码如下。

```
1  private static function _init(){
2      //……
3      //载入函数库
4      require FRAMEWORK_PATH.'function.php';
5  }
```

通过上述代码载入函数库以后，在项目开发时就可以使用 function.php 中定义的函数。

接下来创建项目的配置文件 app\common\config.php，用于保存数据库连接信息，具体代码如下。

```
1  <?php
2  return [
3      'DB_CONFIG' => [
4          'db'   => 'mysql',          //数据库类型
5          'host' => '127.0.0.1',      //服务器地址
6          'port' => '3306',           //端口
7          'user' => 'root',           //用户名
8          'pass' => '123456',         //密码
9          'charset' => 'utf8',        //字符集
10         'dbname' => 'itcast_bxg',   //默认数据库
11     ],
12     'DB_PREFIX' => 'bxg_'           //数据库表前缀
13 ];
```

经过上述修改后，可以在项目中通过“C('DB_CONIG')”来获取数据库配置信息。接下来修改数据库操作类 MySQLPDO.class.php，实现从配置文件中读取配置信息，具体代码如下。

```
1  <?php
2  class MySQLPDO {
3      private static function _connect(){
```

```
4           //获取项目的数据库配置信息
5           $config = C('DB_CONFIG');
6           //准备 PDO 的 DSN 连接信息，连接数据库
7           //……
8       }
9   }
```

在上述代码中，第 4 行将从全局变量获取设置的方式改为通过 C()函数来获取。

4、 基础控制器类

在项目中，由于每一个模块都是一个控制器，多个控制器之间必然会有一些公共的代码，因此可以创建一个基础的控制器类，将公共的基础代码抽取出来。接下来在框架类库目录 framework\library 中创建基础控制器类 Controller.class.php 文件，编写代码如下。

```
1   <?php
2   class Controller{
3       private $_data = [];            //模板变量
4       private $_tips = '';            //提示信息
5       //方法不存在时报错退出
6       public function __call($name, $args){
7           E('您访问的操作不存在！');
8       }
9       //重定向
10      protected function redirect($url){
11          header("Location:$url");
12          exit;
13      }
14      //取出模板变量
15      public function __get($name){
16          return isset($this->_data[$name]) ? $this->_data[$name] : null;
17      }
18      //赋值模板变量
19      public function __set($name, $value){
20          $this->_data[$name] = $value;
21      }
22      //显示视图
23      protected function display(){
24          extract($this->_data);      //将数组转换为变量
25          $this->_data = [];          //释放模板变量
26          require ACTION_VIEW;        //载入视图
27          exit;                       //停止脚本
28      }
29      //提示信息
30      protected function tips($flag=false, $tips=''){
31          $this->_tips = $tips ? ($flag ?  "<div>$tips</div>" :
32          "<div class=\"error\">$tips</div>") : '';
33      }
```

```
34 }
```

在上述代码中，__call()方法用于当调用控制器对象中无法访问或不存在的方法时，提示用户“您访问的操作不存在”；__get()、__set()方法用于类外部访问成员属性时，自动访问内部的$_data 成员属性；display()方法用于载入视图并停止脚本，在载入前会通过 extract()函数将$this->_data 数组转换为变量，用于在视图中使用；tips()方法用于提示信息，第 1 个参数$flag 表示该信息是执行成功时返回的信息还是执行失败时返回的信息，第 2 个参数$tips 是信息的内容。

在创建基础控制器类之后，各功能模块的控制器类都需要继承基础控制器类，示例代码如下。

```
1  <?php
2  //栏目控制器，继承基础控制器
3  class CategoryController extends Controller {
4      //……
5  }
```

5、 基础模型类

在项目中，每个数据表都对应一个模型，多个模型之间会有一些公共代码，可以通过基础模型类抽取这些公共代码。接下来在框架类库目录 framework\library 中创建模型基础类 Model.class.php 文件，编写代码如下。

```
1  <?php
2  //基础模型类，继承数据库操作类
3  class Model extends MySQLPDO {
4      //对 LIKE 条件进行转义
5      public static function escapeLike($like){
6          return strtr($like, ['%'=>'\%', '_'=>'\_', '\\'=>'\\\\']);
7      }
8      //处理 SQL 语句中的 Limit 部分
9      public static function getLimit($page, $size){
10         return ($page-1) * $size . ',' . $size;
11     }
12 }
```

在创建基础模型类之后，各数据表的模型类都需要继承基础模型类，示例代码如下。

```
1  <?php
2  //栏目表的模型类，继承基础模型类
3  class CategoryModel extends Model {
4      //……
5  }
```

当栏目模型类继承基础模型类之后，就可以使用基础模型类中定义的等方法。

任务四：强化模型类

在开发项目时，通常会有大量的数据库操作。为了避免重复的代码书写，可以将一些常见的功能代码抽取出来，提高开发效率。接下来，本任务将在模型类中封装一些常用的数据操作方法。

1、 自动添加表前缀

在为项目数据库中的数据表命名时，使用表前缀是一个好习惯。但是在 PHP 程序开发时，由于需要编写大量的 SQL 语句，当需要修改表前缀时，会带来麻烦。因此，可以通过模型类来实现自动添加表前缀。接下来在基础模型类 framework\library\Model.class.php 中编写 query()函数实现这个功能，具体如下。

```
public function query($sql, $data=[]){
    //SQL 语句模板语法替换（用于自动添加表前缀）
    $prefix = C('DB_PREFIX');
    $sql = preg_replace_callback('/__([A-Z0-9_-]+)__/sU',
    function($match) use($prefix){
        return '`'.$prefix.strtolower($match[1]).'`';
    }, $sql);
    //调用父类（MySQLPDO）执行 SQL
    return parent::query($sql, $data);
}
```

上述代码实现了为 SQL 语句自动添加表前缀。在 query()方法中进行 SQL 语句模板语法替换时，使用了基于回调函数的正则表达式替换函数，表示将正则表达式“/__([A-Z0-9_-]+)__/sU”的匹配结果按照回调函数的返回值进行替换，该正则表达式用于匹配以“__”开始和结束，中间只有一个或多个位于“A~Z、0~9、下划线”范围内的内容。

下面通过代码演示如何实现自动替换表名，具体如下。

```
//模型用法：
$Model = new Model();
$Model->query('SELECT * FROM __CATEGORY__');
//生成 SQL 语句如下：
//SELECT * FROM `bxg_category`;
```

从上述代码可以看出，在开发项目时，表前缀会在模型的 query()方法中自动添加，无需手动编写。经过这样的处理后，当项目需要修改前缀时，只需要在配置文件中修改一次即可，无需改动其它代码。

2、 自动化查询

在实际开发中，对于 SELECT 查询语句的使用非常频繁，因此可以将查询功能进行封装，实现自动生成查询 SQL 语句。在生成 SQL 之前，需要先获知具体操作的表名，因此可以通过模型类的构造方法来传递表名。接下来修改模型类，实现表名的接收，具体代码如下。

```
<?php
class Model extends MySQLPDO{
    protected $table = '';  //保存本模型操作的数据表名
    //通过构造方法接收表名
    public function __construct($table=false){
        parent::__construct();
        $this->table = $table ? C('DB_PREFIX').$table : '';
    }
    //其他方法……
}
```

在获取表名后，接下来在模型类中编写用于查询数据的方法，具体代码如下。

```
//查找数据
```

```
public function select($fields, $data, $mode='fetchAll'){
    $fields = str_replace(',', '`,`', $fields);
    $where = implode(' AND ', self::_fieldsMap(array_keys($data)));
    return $this->$mode("SELECT `$fields` FROM `$this->table` WHERE $where", $data);
}
//根据条件检查记录是否存在
public function exists($data){
    $fields = implode(' AND ', self::_fieldsMap(self::_getFields($data)));
    return (bool)$this->fetchColumn("SELECT 1 FROM `$this->table` WHERE $fields", $data);
}
//将字段数组转换为 SQL 形式
private static function _fieldsMap($fields){
    return array_map(function($v){ return "`$v`=:$v"; }, $fields);
}
```

在上述代码中，select()方法用于根据$data 传递的查询条件，查询$fields 字段，以$mode 方法进行结果集处理；exists()方法用于根据$data 查询条件，判断查询记录是否存在。

下面通过代码演示如何使用 select()和 exists()方法，具体如下。

```
//模型用法：
$Category = new Model('category');
$Category->select('name,pid', ['id'=>1], 'fetchRow');
$Category->exists(['id'=>1]);
//生成 SQL 语句如下：
//SELECT `name`,`pid` FROM `bxg_category` WHERE `id`=1;
//SELECT 1 FROM `bxg_category` WHERE `id`=1;
```

从上述代码可以看出，通过强化后的模型类可以快捷地进行数据查询。

3、 自动化数据操作

在项目中，对于数据的添加、修改和删除也是经常会用到的操作。因此接下来在模型类中实现自动化数据库操作的方法，具体如下。

```
//添加数据（支持批量添加）成功返回最后插入的 ID，失败返回 false
public function add($data){
    $fields = self::_getFields($data); //获取所有字段
    $sql = "INSERT INTO `$this->table` (`".implode('`,`', $fields).'`)
    VALUES (:'.implode(',:', $fields).')';
    return $this->query($sql, $data) ? $this->lastInsertId() : false;
}
//修改数据（支持批量修改）
public function save($data, $where='id'){
    //获取所有 WHERE 字段
    $where = explode(',', $where);
    //获取所有操作字段
    $fields = array_diff(self::_getFields($data), $where);
    $fields = implode(',', self::_fieldsMap($fields));
```

```
15      $where = implode(' AND ', self::_fieldsMap($where));
16      return $this->exec("UPDATE `$this->table` SET $fields WHERE $where", $data);
17  }
18  //修改单个字段
19  public function change($field, $old, $new){
20      return $this->exec("UPDATE `$this->table` SET `$field`=:new
21      WHERE `$field`=:old", ['new'=>$new, 'old'=>$old]);
22  }
23  //删除记录（支持批量删除）
24  public function delete($data){
25      $fields = implode(' AND ', self::_fieldsMap(self::_getFields($data)));
26      return $this->exec("DELETE FROM `$this->table` WHERE $fields", $data);
27  }
28  //自动从一维或二维数组中获取字段
29  private static function _getFields($data){
30      $row = current($data);
31      return array_keys(is_array($row) ? $row : $data);
32  }
```

上述代码实现封装了数据库的添加、修改和删除操作，其中 add()、save()、delete()方法支持批量操作。在实现批量操作时，为参数$data 传递二维数组即可。save()方法的第 2 个参数表示 WHERE 中的字段，当指定后，$data 数组中的相应元素将作 WHERE 中的字段。change()方法用于修改单个字段，参数$field 表示待修改的字段，$old 表示该字段修改前的值，$new 表示修改后的值。

下面通过代码演示自动化数据操作方法的使用，具体如下。

```
1  //实例化模型
2  $Category = new Model('category');
3  //自动插入一条栏目数据
4  $Category->add(['name'=>'Java', 'pid'=>0]);
5  //自动插入多条栏目数据
6  $Category->add([
7      ['name'=>'Android', 'pid'=>0],
8      ['name'=>'MySQL', 'pid'=>0]
9  ]);
10 //自动修改 id 为 1 的栏目的名称
11 $Category->save(['id'=>1, 'name'=>'HTML'], 'id');
12 //将 sort 字段所有值为 2 的记录改为 0
13 $Category->change('sort', 2, 0);
14 //删除 id 为 2 的栏目
15 $Category->delete(['id'=>2]);
```

4、 自动实例化模型

在控制器中实例化模型时，当多个方法用到同一个表的模型时，就会出现重复的实例化操作。为了解决这个问题，可以编写函数来将实例化的模型通过静态变量进行保存，同时还可以判断自定义的模型类是否存在。接下来在 framework\function.php 函数库中编写函数，实现模型的实例化，具体代码如下。

```
1  //实例化特定表的模型
```

```
function D($name){
    static $Model = [];
    $name = strtolower($name);
    if(!isset($Model[$name])){
        $class_name = ucwords($name).'Model';
        $Model[$name] = is_file(MODEL_PATH."$class_name.class.php") ?
        new $class_name($name) : new Model($name);
    }
    return $Model[$name];
}
//实例化空模型
function M(){
    static $Model = null;
    $Model || $Model = new Model();
    return $Model;
}
```

上述代码定义了两种实例化模型的函数，其中 D()函数用于实例化特定表的模型，M()函数用于实例化基础模型。在调用函数时，可以传递数据表的名称作为参数，从而让模型类获知待操作的表名称。

在完成模型实例化的函数后，下面通过示例代码进行演示，具体如下。

```
//先实例化后调用
$Category = D('category');
$Category->select('name', ['id'=>1]);
//实例化后直接调用
D('category')->select('name', ['id'=>1]);
//实例化基础模型
M()->query('SELECT * FROM __CATEGORY__');
```

从上述代码可以看出，D()函数和 M()函数简化了模型实例化的代码，并且当函数多次调用时，同一个表的模型只实例化一次。