Web 开发技术丛书

PHP 与 MySQL 高性能应用开发

杜江　著

机械工业出版社
China Machine Press

图书在版编目（CIP）数据

PHP 与 MySQL 高性能应用开发 / 杜江著 . —北京：机械工业出版社，2016.8
（Web 开发技术丛书）

ISBN 978-7-111-54796-9

I. P…　II. 杜…　III. ① PHP 语言－程序设计　② 关系数据库系统　IV. ① TP312
② TP311.138

中国版本图书馆 CIP 数据核字（2016）第 214629 号

PHP 与 MySQL 高性能应用开发

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：李　艺　　责任校对：董纪丽
印　　刷：　　版　　次：2016 年 9 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：19.25
书　　号：ISBN 978-7-111-54796-9　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

Preface 序

曾经我与你一般，年少时期，对人生只知努力，却不知何往，只得上下求索，东寻西觅。于是求知识、读文字、写代码、做架构，时至而立之年方初识端倪。几年来亲历创业，一路走来有技术的积累，亦有技术外的磨砺。比照更多的同路者，做自己最擅长的才更有力量。

当今社会，如你我这样依靠技术成就理想的开发者，共同特征是吃苦耐劳，也有一些完美主义。我们在互联网上获取大量知识，而上面的信息多数可受其益，但陈旧错漏之文仍有，条理逻辑亦差强人意，难免蒙受其弊。因此，纸质图书阅读对于开发者来说仍有必要。

开发类书籍创作大都不是轻松的工作，但我心中一直存有一份责任，那就是让更多的朋友能够解惑并能目标明确地向前，让“Open & Share”的开源理念得到更多理解，这也是我能够坚持的初心。

每晚在称为“中国硅谷”的中关村软件园区，从窗外看着外面灯火通明的百度大厦，还有很多人在加班工作。也有很多技术类的创业者，他们都在执着地用自己的双手浇灌未来的理想之花。每当此时，耳畔听着西山风声，手中的键盘声响起，眼前屏幕的文字跃动，是另一种喜悦。

创新来源于每天的思考与实践，梦想方能不绝于缕。互联网的新技术每天都在发展，关于 LAMP/LNMP 开发、高性能、高扩展的话题也一直在更新发展中。

本书持续写了两年有余，其中针对 PHP 升级，部分内容也同步做了更新，特别是 PHP7 的发布。书中内容符合 PHP5.6 及以上版本。希望本书能够帮助你避免在开发时遇到坑，或者简单问题复杂化，进而提高编码效率。

人生处处是战场，作为开发者的我们，需要每日积跬步行千里，不断实践让自己更加优秀。既然你已经准备好了，就让我们充满感激和动力，出发！

杜江（别名：洛逸）

前　言 Preface

在过去的十几年间，LAMP 开源技术推动着互联网开发，有 4000 万以上的网站在使用 LAMP&LNMP 技术平台驱动。

在互联网和移动互联网平台中，其中 Facebook、开心网、新浪网、Yahoo!、百度、腾讯、搜狐、网易及各个视频网站全部或大部分使用的是 LAMP&PHP 技术。

与其说 Web 的伟大创新，不如说是创新者的智慧，还有 PHP 技术的鲁棒性与相对于其他语言的快速、灵活、敏捷性，给互联网——这个亦庄亦娱的行业带来强大的动力。

近年来，PHP 与互联网正一起创造着流行。2000 年前后，PHP 应用于 Yahoo! 网站，国内门户网站腾讯、新浪、优酷、凤凰及众多在线网络游戏厂商等也都全部或部分使用 PHP 技术。同时，PHP 也为互联网的新兴网站创造了一个又一个神话。

Craigslist.org 是在全美第 6 名、全球第 20 名的分类信息网站，每月有 1000 万独立访问量和 30 亿页面浏览量，它使用 LAMP 技术开发，国内类似的网站如赶集网、百姓网也全部使用 PHP 技术。

维基百科（Wikipedia），也称为自由的百科全书。它是由全球不同民族、不同语言共同编撰的一部网络百科全书，由 PHP 开发，并以 Mediawiki 开放源代码。

Yelp 是美国最大的店铺点评网站，相当于中国的大众点评网，2009 年婉拒了 Google 近 6 亿美元的收购要约，目前已成为消费者购买与体验商品的最佳社区，国内有安居客、蚂蚁、小猪短租、好车无忧等类似网站也全部使用了 PHP 技术。

SNS（Social Networking System）巨头 Facebook，是全球最大的 LAMP 网站，目前已有超过 15 亿用户，超过 Google。目前这个全球最火热的社区，已演化为人们生活不可缺少的工具。国内类似的 SNS 网站，如开心网、同学网、腾讯朋友等全部使用 PHP 开发。而 Facebook 的社交开发商（Social Game Developer），如 Zynga 等社交游戏厂商也应

用了 PHP 开发，因为 Facebook 的巨大应用量而赚得盆满钵满。

随着 Twitter 的流行，使国内微博网站愈加火爆，如新浪微博、腾讯微博等网站全部使用了 PHP 开发。而热门、模式创新的网站，非 Foursquar.com 和 Groupon.com 莫属，它们分别是基于位置的地图服务和团购商品的服务，而这些网站的中国版如美团、团宝等网站使用的也是 PHP 技术。

PHP 在电子商务 / 社交化电子商务领域，以及企业软件上同样大展身手，如淘宝前端使用 PHP、Prestashop、ShopEx、Magento、eCart、osCommerc 等。可以预见的是，在未来还会有新的互联网神话出现，而加速这些网站前进的 PHP 将继续担当主力。

还有企业级开发领域，如 Zend、SugarCRM、DotProject 等，也在使用 PHP 来实现云计算等企业级开发领域。而且在当今如火如荼的移动互联网以及网页游戏开发领域，还有 PHP for Android 等框架来帮助开发者实现本地化 App 开发的想法，而且 App 的后面也可使用 PHP 来提供 API 服务接口。

PHP 并非万能，但凭借它实用高效的优势，在 Web 开发领域，PHP 和 MySQL 无疑是“世界上最好的语言”。

现今，国内的各个互联网公司均面临两大问题和挑战：第一，高流量、高负载的商务应用使 Web 系统不堪重负；第二，价格高昂的带宽、硬件、商业软件等成本高居不下，越来越多的互联网公司开始拥抱开源的 LAMP/LNMP 平台。

同时，PHP 也在不断更新。我们需要有众多热爱编程开发，有扎实的基础以及丰富的实际编程经验，有创新、有思想的工程师，加入到 PHP 开发的行列中。

为什么要使用本书

如果你已经看过市场上很多初级类书籍，却还在寻找 PHP 编程思想、底层原理、编程技巧、可伸缩性、可靠性、开发规范等内容，那么就请使用本书，相信可以获取更多新鲜与深入的主题。

本书为读者带来的是一系列实用的、进阶的“干货”，相信定会给你的程序生涯和未来发展带来帮助。

书中主要介绍如下主题：

- ❑ 解惑：掌握 PHP 编程中的“长尾”细节。
- ❑ 深入：PHP 面向对象高级开发。
- ❑ 浅出：PHP 开发中的调试与技巧。

- ❑ 编程之道：透彻理解面向对象开发思想与设计模式。
- ❑ 更快：使用 OpCode 缓存。
- ❑ 扩展：memcached 及扩展应用。
- ❑ 搜索：Sphinx 全文搜索引擎。

为了提供更好的实用性，本书除了详解 PHP 中的深度开发外，还提供了相应的代码实例。读者可登录 21CTO（www.21cto.com）本书相关页面下载。

本书写给谁

本书适合 PHP 中级开发及以上资质的读者，需要读者充分了解 PHP 技术，可结合其他书籍进行同步阅读。

本书读者对象可为 PHP 研发工程师、软件架构师、系统架构师。本书也可作为 IT 运维人员、DBA、计算机专业本科以上学生的参考用书。

本书特点

书中讲解了 PHP 5.6 以上及 PHP7.02 版本的新特性，涵盖了目前大中型网站使用的研发技术，包括扩展、伸缩、负载、优化等，以及实际研发中的解决方案。本书不只停留在代码应用层，还包括架构方面的方法与思路，相信会帮助读者更好掌握 PHP。

致谢

感谢机械工业出版社杨福川、高靖雅和李艺，以及曾经并肩战斗的朋友，是你们的鼓励才能使本书得以展现给各位。PHP 由 PHP 开发小组和众多的 PHPer 共建。同样，本书也得到了很多同仁的支持，在此一并致谢！

社区支持

如果你从本书中发现错误或漏洞，或者发现一些有价值和感兴趣的内容，可登录本书的技术支持平台：21CTO（www.21cto.com）与笔者进行交流。

同时，欢迎大家提出宝贵意见，以便在本书再版时为读者带来更好的体验。

Contents 目　录

第 1 章 Chapter 1

PHP 解惑

和其他语言相比，PHP 给人的印象是入门简单的语言。当你的技术能力达到一定阶段时，会发现情况并非如此。PHP 采用“极简主义”，就是以入门容易为准则设计的，在十几年的持续发展历程中，它早已成为一个开源领域的语言且具备现代语言特性的平台之一，在 Web 开发领域，我们相信 PHP 就是“世界上最好的语言”。

人无完人，语言也一样。天下事物都需要花大量精力去研究实践，深入下去不是易事，了解越多越敬畏。况且 Web 开发又是个严谨创意，如不能通透理解隐藏在后面的深层机制，就有可能损害应用的性能，导致低级错误的发生。

互联网产品的特性是小步快跑，快速迭代。这就经常需要我们直接开发，为快速实现功能而忽略一些性能、降低代码质量，但上线后一定要对代码进行整理、优化与修正。事实上，有的开发者从事开发若干年，却未必会对一些技术原理深究，加上网上大量的开源代码，借 Google、Github 等发扬拿来主义，复制粘贴未经推敲的代码，似乎没花太大力气就完成了任务。由于不同的架构设计，没有经过严谨的代码审核，这样的代码怎么能保证产品正常运行？

古人有这样一句话——“勿以浮沙筑高台”，即不要在浮沙上面建筑高台。基础不扎实，台子搭得再高也会倒掉，没有坚实的基础，是无法做好开发的。为保证开发的网站平台健壮，使平台能够承载更高的流量，需要理解、领悟更多的技术点，才能写出高质量、高扩展、高性能的代码。

1.1 省略结束标签的便利性

一个优秀的程序员会在编码前习惯把 PHP 标签成对写完，再写功能逻辑——我也不例外，不过有一次忘记了写结束标签，却发现也能正常运行，当时感觉很奇怪，还以为是神奇的 PHP 高度容错的结果。

其实对于 PHP 编译器来说，脚本的结束标签“ ?>”是可选的，在写程序时你可以忽略它。你或许碰见过：在使用 include()、require() 或输入输出缓冲函数时，页面顶部有时多空行或者出现“ header had send”之类的错误信息，这类问题与结束标签有关。省略结束标签适合纯 PHP 文件，如果是 PHP 与 HTML 混合开发，则不可省略。

忽略结束标签不仅能少写两个字符，还让我们的开发更顺利，何乐而不为。

1.2 emtpy、isset、is_null 的区别

变量在所有计算机语言中均有提供，它用来保存数值、文本、对象等内容。我们可以把变量看作一个有名称的桶，里面放着一个值，这个值可以是数字、字符串或对象，以及包含你想到的任何合法的内容。

PHP 提供了 3 个用于测试变量值的函数，分别是：isset()、empty() 和 is_null()。这几个函数均返回布尔值，有时使用不当会造成意想不到的结果，需要详细说明。

比如，用 isset() 和 empty() 返回的结果是相反的，但有时却并非一直如此，下面我们一起来了解这几个函数的具体区别。

isset() 用来检测一个变量是否已声明且值不为 NULL。换句话说，只能在变量值不是 NULL 时返回真值。

empty() 用来检测一个变量是否为空，也就是说有如下情况时返回真值：变量是一个空字符串，false，空数组 [array()], NULL，0，' '，以及被 unset 删除后的变量。

在 PHP5.5 之后，empty() 函数可以接受任意类型的表达式。

正确地检查一个变量是否为空，可使用如下格式：

```
if(empty($approve)){
    //etc
}
```

这种形式可适用在 PHP 的任意版本中。如果你用的是 PHP5.5 以上版本，可以使用

如下格式：

```
if(empty(0)){
    //etc
}
if(empty(CreateNew())){
        //etc
}
```

以上格式在 PHP5.5 以上版本中均可以使用，如果小于该版本会返回解析错误。

is_null() 函数用来判断变量内容是否是 NULL 值，即返回真值的条件仅为变量是 NULL 时。值得一提的是，is_null() 是 isset() 函数的反函数，区别是 isset() 函数可以应用到未知变量，但 is_null() 只能针对已声明变量。

我们用一张表格来汇总这些函数返回值的不同之处（表 1-1），表中空白表示函数返回布尔值假（false）。

表 1-1　测试函数返回值的区别

对比项			
变量值（$var）	isset($var)	empty($var)	is_null($var)
" "（一个空字符串）	bool(true)	bool(true)	
" "（空格）	bool(true)		
FALSE	bool(true)	bool(true)	
TRUE	bool(true)		
array()（一个空数组）	bool(true)	bool(true)	
NULL		bool(true)	bool(true)
"0"（0 是一个字符串）	bool(true)	bool(true)	
0（0 是一个整型值）	bool(true)	bool(true)	
0.0（0 是一个浮点值）	bool(true)	bool(true)	
var $var;（一个变量声明，但是没赋值）		bool(true)	bool(true)
NULL byte（'\ 0'）	bool(true)		

1.3　布尔值的正确打开方式

关于布尔值，在 PHP 中可以这么来写：

```
<?php $flag = True; ?>
<?php $flag = TRUE; ?>
```

```
<?php $flag = true; ?>
```

有点儿像孔乙己的“茴香豆”写法，这 3 段代码都可以正常运行。但是，哪个最好？哪个是正确的？在 PHP 中，常量规定为大写，第二行代码显然是正确的。

下面我们再来看一下比较语句。比较常用于两个变量之间，但是，也会有这样的代码：

```
<?php
if($price = $cart->price){
    echo 'function return TRUE';
}else{
    echo 'function return FALSE';
}
?>
```

可以看到，这段代码也没有错，但不怎么容易理解。仔细看，这个分支里面的表达式是一个变量跟一个对象方法的赋值，并不是一个布尔值运算，很容易把人引入不正确的思路。

这种方法尽量不要用。正确的写法可以是这个样子的：

```
$user_id == $user->getUserId()
```

1.4 变量作用域实践

我们知道，在 PHP 中定义一个变量后，在脚本任意位置都可以存取访问，这被称为“全局变量”，而定义在函数或类的方法中的变量只可以在函数内部访问，这叫作“局部变量”。

使用局部变量可以使源代码易于管理，试想如果所有的变量都是全局的，任何位置都可访问、修改它的内容，如果变量重名就可能发生“污染”。通过声明局部变量来限制一个变量的存取范围，可以让代码模块化，易调试，让应用运行更健壮。

下面我们就来看看如何使用全局变量和局部变量，如代码清单 1-1 所示：

代码清单1-1　使用全局变量与局部变量

```
<?php
$globalName = "老杜";
function getvar() {
    $localName = "Raymond";
    echo"Hello, $localName!<br>";
```

```
}
getvar();
echo "The value of \$globalName is: '$globalName'<br />";
echo "The value of \$localName is: '$localName'<br />";
?>
```

该脚本运行后将显示如下内容：

```
Hello, 老杜!
The value of $globalName is: 'Raymond'
The value of $localName is: ''
```

在上面的代码中，我们一共创建了两个变量：其中 $globalName 是全局变量，它没有在任何函数体里；另一个是名为 $localName 的局部变量，是在 sayHello() 函数里内部定义的。

程序运行时先是调用 sayHello() 函数，显示的是“hello,Raymond！”，接下来用 echo 显示两个变量，分别是 $globalName 和 $localName。由于 $globalName 是定义在函数之外的全局变量，在脚本任何位置都可以访问，因此显示为“ Raymond”。而 $localName 定义在 sayHello() 函数内部，只能在函数内访问。脚本中使用 echo 来访问这个局部变量，而 PHP 不允许外部访问此局部变量。因此运行时，PHP 认为程序要创建一个新的全局变量 $localName，并将默认值初始化为空，所以显示的时候是空白的。

PHP 允许函数内部可访问外部全局变量，只需在函数中使用 global 关键字即可。我们来看代码清单 1-2：

代码清单1-2　使用全局变量与局部变量

```
<?php
$globalName = "老杜";
function sayHello() {
    $localName = "Harry";
    echo "Hello, $localName!<br />";
    global $globalName;
    echo "Hello, $globalName!<br />";
}
sayHello();
?>
```

该段脚本会输出下面的内容：

```
Hello, Harry!
Hello, 老杜!
```

由于在 sayHello() 函数里使用了 global 来声明 $globalname 为全局性质，因此它的内容被打印了出来。

1.5 多维数组排序

使用 PHP 开发应用，几乎就是一直跟数组打交道。PHP 数组的强大和灵活性能够解决大部分应用的问题。在数组编程中，常用的有 sort()、ksort() 等相关函数，使用它们就可以很方便地处理一维数组，比如按键值降序和升序排列。

这些函数不能用于多维数组，但是在开发中常常是对多维数组排序处理。下面我们定义一个二维数组，如代码清单 1-3 所示：

代码清单1-3 定义一个标准二维数组

```
<?php
$a = array(
        array("sky", "blue"),
        array("apple", "red"),
        array("tree", "green")
    );
?>
```

这是一个简单的二维数组，数组的元素也是数组。我们可能需要对 userid 这个键排序，或者按汉字或英文字符排序。

为了给多维数组进行排序，我们需要自定义排序函数，然后再调用 sort()、usort()、ksort() 这些函数，让这些函数使用自定义函数。

uasort 函数接受两个参数，并且返回一个值表示哪个参数应该排在前面。负数或 FALSE 意味着第一个参数应该排在第二个参数之前。正数或者 TRUE 表示第二个参数应该排在前面，如果值为 0，则表示两个参数相等。

下面，我们对前面的数组第一个键进行排序，代码清单 1-4 是一个自定义函数。

代码清单1-4 将数组按键值排序的自定义函数

```
function my_compare($a, $b) {
if ($a[1] < $b[1]) {
        return -1;
    }else if ($a[1] == $b[1]){
        return 0;
    }else{
        return 1;
```

```
    }
}
```

这样一来，我们可以后面使用 uasort 调用这个自定义函数：

```
uasort($a, 'my_compare');
```

PHP 会把内层数组不断地发送给此自定义函数，从而将它排序完成。想要了解排序细节，可以输出函数里被比较的数值，由此我们可以看出自定义排序是如何被调用的。代码清单 1-5 是脚本的完整代码。

代码清单1-5　多维数组排序

```
<?php
//定义多维数组
$a = array(
        array("sky", "blue"),
        array("apple", "red"),
        array("tree", "green")
    );
//自定义数组比较函数，按数组的第二个元素进行比较
function my_compare($a, $b) {
    if ($a[1] < $b[1]) {
        return -1;
    }else if ($a[1] == $b[1]){
        return 0;
    }else{
        return 1;
    }
}

//排序
usort($a, 'my_compare');
//输出结果
foreach($a as $elem) {
    echo "$elem[0] : $elem[1]<br />";
}
```


1.6　超级全局数组

超级全局数组（super global array）是由 PHP 内置的，无须开发者重定义。PHP 执行时会自动将当前脚本需要收集的数据分类保存在这些超级全局数组中，这些数组有十多

个分类，每个数组保存的内容和功能不同，如表 1-2 所示。

表 1-2　超级全局数组的分类与功能

名　　称	功　　能
$_GET[]	取得用 GET 方法提交的表单内容，数组键和值分别对应元素名和值
$_POST[]	取得用 POST 方法提交的表单内容，数组键和值分别对应元素名和值
$_COOKIE[]	取得或设置当前站点的 Cookie
$_SESSION[]	取得当前用户访问的会话，以数组形式体现，如 sessionid 及自定义 session 数据
$_ENV[]	当前 PHP 服务器的环境变量
$_SERVER[]	当前 PHP 运行环境的服务器变量
$_FILES[]	用户上传文件时提交到当前脚本参数
$_REQUEST[]	包含当前脚本提交的所有请求，它包含了 $_GET、$_POST、$_COOKIE、$_SESSION 这些超级全局数组的全部内容
$GLOBALS[]	该超级变量数组包含正在执行脚本时所有超级全局数组的内容

$GLOBALS 超级全局数组可以让我们在函数里访问全局变量，如代码清单 1-6 所示：

代码清单1-6　在函数中访问外部变量

```
<?php
$globalName = "我是全局变量";
function sayHello() {
    echo"你好, " . $GLOBALS['globalName'] . "<br / >";
}
sayHello();  // 将显示 "你好, 我是全局变量"
?>
```

1.7　global 关键字与 global 数组的区别

你也许记得，前面我们提到过 global 关键字和 global 数组。那么问题来了，它们长得如此像，似乎功能也相同，到底有什么区别？我们分别来看一下。

$GLOBALS['var'] 是外部的全局变量本身，global $var 是外部 $var 的同名引用或者指针，如代码清单 1-7 所示：

代码清单1-7　删除全局变量

```
<?php
$var1 = 1;
```

```
function test(){
    unset($GLOBALS['var1']);
}
test();
echo $var1;
?>
```

因为 $var1 变量被删除，所以没有内容显示出来。请再看如下代码：

```
<?php
$var1 = 1;
function test(){
    global $var1;
    unset($var1);
}
test();
echo $var1;
?>
```

此段代码意外地打印了 1。这是为什么？因为删除的只是个别名引用，其本身的值并没有任何更改。

global $var 与 &$GLOBALS['var'] 等价，相当于调用外部变量的一个别名，所以上面代码中的 $var1 和 $GLOBALS['var1'] 指向的是同一个变量。

PHP 的全局变量和 C 有一点点不同。在 C 语言中的全局变量在函数体内无效。而在 PHP 中，在函数中想调用外部全局变量时可用 global 声明。PHP 的“全局”不是指整个网站，而是应用于当前页面，包括 include 或 require 的全部文件。

综合以上内容，我们总结出如下结论：

- $GLOBALS['var'] 是外部的全局变量本身。
- global $var 是外部 $var 的同名引用或者指针。

1.8　活用静态变量

在 PHP 脚本函数内部创建的局部变量，执行时是存在的，当执行完毕后会在内存里立即删除，再次运行函数时会重新创建。这样的优点是：确保函数每次执行是完整独立的，以免混乱。

但我们有时会想在函数调用时保存上次局部变量执行的结果，以便下次执行时使用，这时就可以用静态变量来实现。

声明一个静态变量只需在函数体中变量前面加入关键字 static 声明，并初始化一个值，如代码清单 1-8 所示：

代码清单1-8　使用静态变量

```
<?php
function myFunction() {
    static $myVariable = 0;
}
?>
```

通过一个实例比较静态变量是如何有用的，我们先编写一个自定义函数，它的功能是返回函数被调用的次数。如代码清单 1-9 所示：

代码清单1-9　函数内值的累加

```
<?php
function createWidget(){
    $numWidgets = 0;
    return++$numWidgets;
}
echo "Creating some widgets...<br />";
echo createWidget() . " created so far.<br />";
echo createWidget() . " created so far.<br />";
echo createWidget() . " created so far.<br />";
?>
```

这段代码执行后结果如下：

```
Creating some widgets...
1 created so far.
1 created so far.
1 created so far.
```

我们三次调用 createWidget() 函数，每一次函数被调用时，内部的 $numWidgets 变量都会从 1 开始，而不是每次累加，没有达到想要的预期结果。

而使用静态变量，就可以在每次函数调用时使用它上次运算的值。下面修改一下代码，将局部变量声明为静态变量，如代码清单 1-10 所示：

代码清单1-10　使用函数内使用静态变量累加

```
<?php
function createWidget(){
    static $numWidgets = 0;
    return ++$numWidgets;
}
```

```
echo "Creating some widgets...<br / >";
echo createWidget() . " created so far.<br / >";
echo createWidget() . " created so far.<br / >";
echo createWidget() . " created so far.<br / >";
?>
```

现在，脚本会输出我们预想的结果：

```
Creating some widgets...
1 created so far.
2 created so far.
3 created so far.
```

综上所述，静态变量在函数调用时，保存了上次运行的值。当脚本运行完毕退出时，静态变量也会销毁，这一点和全局、局部变量特性相同。

1.9　require、require_once、include、include_once 与 autoload

这是一个老问题。我们先进入一个情境，在写代码时，我们会把刚写完的函数或类归并到不同的文件中，根据功能把这些文件保存在某个目录里，再使用 include()/include_once() 或 require()/require_once() 包含它们来执行，以提高代码的重用性和简洁性。

这 4 个函数在产品代码里可以交替使用，但在性能上有着一些细微差别，特别是对性能要求高的项目，需要花点儿心思来分析。

include()（中文意为包含）和 require()（中文意为必须）允许在当前脚本中多次执行包含的文件。include_once() 和 require_once() 确保在执行时对包含的文件只执行一次，即使在代码中调用很多次。

如果 PHP 虚拟机在 PHP 脚本中扫描到 include() 或 include_once() 语句，若包含文件失败，会显示警告错误（Warning Error），然后还会继续执行。如果是 require() 或 require_once() 语句，包含文件失败后会抛出致命的错误提示（Fatal Error）并且中止脚本的执行。

这样当文件意外丢失或者逻辑（比如重复性包含、函数重命名等）出错时，想要脚本继续执行并输出页面，我们可以使用 include() 或 include_once()。

在开发一个严谨应用时，要使用 require() 或 require_once() 来进行包含操作，即便包含的不是 PHP 文件，这样有利于应用程序的安全、完整以及健壮性。

错误只会在一个文件未被包含时触发，多数是目录存取权限或路径写错这些原因引

起的。在实际的运营环境里，注意千万别把程序的错误信息抛给用户，可在代码中使用 error_reporting(0) 禁止所有的错误显示，内部加入完善的错误与日志处理，只给用户显示正常的内容。

从性能角度考虑，由于在导入 PHP 脚本时进行大量的操作状态（stat）调用，使用 require() 要快于 require_once()。比如你请求包含的文件都在 /var/www/myapp/modules/myclasss.php 下，则操作系统会在到达 myclasss 之前的每个目录进行一次 stat 调用，如代码清单 1-11 所示：

代码清单1-11　使用require_once()包含文件

```
<?php
require_once('ClassA.php');
require_once('ClassB.php');
require_once('ClassC.php');
require_once('ClassD.php');
echo '测试require_once';
?>
```

它们一共用了 4 个文件，均通过 require_once() 请求并包含。它们所请求的类均在以下代码中，它们仅声明了自己，并没有包含任何函数实现，如代码清单 1-12 所示：

代码清单1-12　声明要包含的类

```
<?php
class A{
}
class B{
}
class C{
}
class D{
}
?>
```

以上 4 个空类可以帮助我们模仿一个需要在主脚本中使用外部 PHP 文件的 PHP 脚本。

我们排除了任何额外的函数调用，专注于使用 require_once() 函数的文件加载。把每个类分别放于一个单独的文件中，命名为 ClasssA.php、ClassB.php、ClassC.php、ClassD.php，与代码清单 1-10 放在同一个文件夹下。

Apache 为我们提供了 ab（apache benchmark）工具，可以用它来测试使用 require_

once() 的脚本性能：

```
ab -c 10 -n 100000 localhost/index.php
```

本例中我们模拟了 10 万个请求，同一时间有 10 个并发请求，结果如图 1-1 所示。

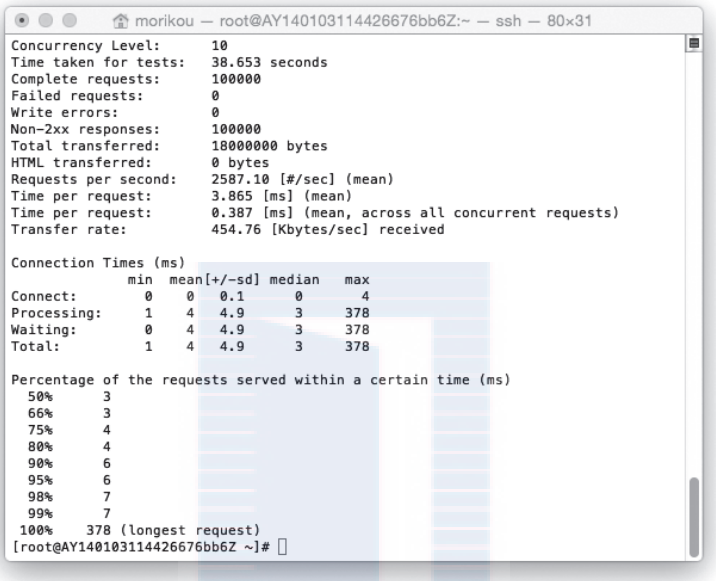

图 1-1　使用 ab 测试并发请求结果

使用 ab 工具测试 require_once()，可以看到响应时间为 38.653 ms，另外结果还显示，这个脚本每秒可以支持 2587.10 个请求。

现在我们将 require_once() 改为 require()，如代码清单 1-13 所示：

代码清单1-13　使用require()包含文件

```
<?php
require('ClassA.php');
require('ClassB.php');
require('ClassC.php');
require('ClassD.php');
echo '测试require_once';
?>
```

重新启动 Web 服务器，再运行刚才的 ab 测试命令，结果如图 1-2 所示。

测试结果表明，使用 require() 后，每秒可以支持的请求数量有所提升，从 2587.10 提升到 2589.38。此结果还说明，代码的响应时间从上面的 38.653 ms 降到了 38.619 ms，

减少了 2 ms。

如果想自动加载某个文件，可以用类似下面的 autoload 函数，如代码清单 1-14 所示：

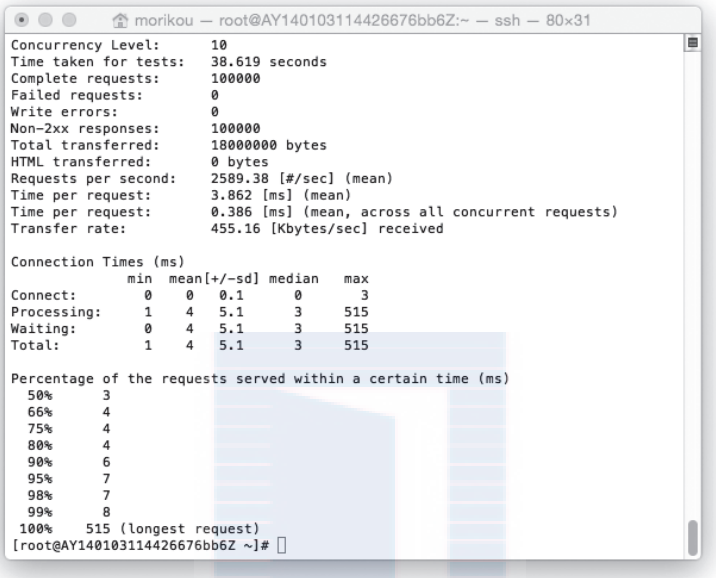

图 1-2　使用 ab 测试 require() 函数的性能

代码清单1-14　使用autoload函数自动引用

```
spl_autoload_register('autoload');
function _autoload($name) {
    require('include/' . $name . '.php');
}
```

我们用 spl_autoload_register 告诉 PHP 在执行时自动加载某函数，在这里我们告诉它调用 autoload 函数，autoload 函数根据需要的文件，使用 require 来包括相关的文件。

可以确定的是，require_once 要慢于 require，使用 autoload 速度最快。另外，在代码中函数调用越少，它的运行速度就越快。

1.10　= 与 ==、=== 的区别

＝＝和＝＝＝都是比较运算符，用来处理两个操作数之间的关系操作。

当操作数是两个字符串时，按 ASCII 字符顺序比较；当操作数是数字时，按数字大小比较，比较后返回一个布尔值 true 或 false。

我们通过代码清单 1-15 来做对比：

代码清单1-15　＝＝与＝＝＝的表达式

```
<?php
$x = 23;
// PHP自动把字符串转换为整型数据
echo ($x == 23)  .  " <br / > "; // 显示 1 (true)
echo ($x === 23 ) . "<br / > "; // 显示 1 (true)
echo ($x === "23 ")  . " <br / > "; // 显示为空 (false)
?>
```

我们看到，第 4 行代码使用了＝＝＝全等比较，因为它后面的 23 是显式声明为字符串，两侧数据类型并不一致，因此返回布尔值 false。

其实我们着重讲解的是＝＝和＝＝＝的区别，而＝是一个等号，它是一个赋值操作符，即把等号右边的值赋值给左侧的变量。

1.11　HereDoc 与 NowDoc

PHP 的 HereDoc 以 Linux 系统的“原型文档”（here-document）语法为基础，它允许开发者在脚本中嵌入一段文本内容，如邮件模板、短信模板、HTML/JavaScript 脚本等。

它是一种面向字符行的引用，所以定界符是针对行而不是字符，起始定界符是当前行，结束定界符是一个指定字符的行。如代码清单 1-16 所示：

代码清单1-16　HereDoc之声明

```
<?php
echo<<<THIS_HEREDOC
    PHP stands for "PHP: Hypertext Preprocessor".
    The acronym "PHP" is therefore,  usually referred to as a recursive acronym
because thelong form contains the acronym itself.
THIS_HEREDOC;
?>
```

可以看到这里以 <<<THIS_HEREDOC 开头，以 THIS_HEREDOC 结束，表示引用结束。

在 <<< 后面的名字可以是任何你喜欢的名称，比如可以用“HELLO”定义 HereDoc 开始，然后末尾就以“HELLO”表示 HereDoc 语句的结束。

在 HereDoc 中可以直接引用 PHP 变量（前提是该变量已经定义），HereDoc 会解释该变量，直接显示该变量的值，为避免与其他文字混淆，可以用花括号将该变量括起来。

如代码清单 1-17 所示：

代码清单1-17　在HereDoc中引用PHP变量

```
<?php
$output = "LAMP高性能开发";
$content = <<<THIS_HEREDOC
实践{$output}
THIS_HEREDOC;
?>
```

注意，在使用 HereDoc 时，结束符之前的文字要与上面文字段落自然换行，不要用空格或 TAB 字符进行缩进操作，否则 PHP 会提示解析错误。

如果想在 HereDoc 内容中显示 $ 打头的字符串，PHP 会认为是一个变量，因为不是合法的标识会提示编译错误，需要进行转义操作。

如果我们使用 PHP 5.3 以上版本，就可以使用新的语法 NowDoc 了。它相当于 HereDoc 中内容都自动转义，文本中的内容包括变量都不会解析。

NowDoc 源于 HereDoc，语法和 HereDoc 相似，唯一区别是它使用单引号作为定界符，如代码清单 1-18 所示：

代码清单1-18　NowDoc的使用

```
<?php
$religion = 'Hebrew';
$myString = <<'END_TEXT'
"'I am a $religion,' he cries - and then - 'I fear the Lord the God of
Heaven who hath made the sea and the dry land!'"
END_TEXT;
echo "<pre> $myString</pre>";
?>
```

NowDoc 对包含的文本均不做任何解析。在输出时，里面的内容都当作纯文本，无论有没有 PHP 变量还是特殊字符，这非常适合文本中含有代码的内容，比如想在脚本中显示一段 PHP 源码、动态 SQL 语句等具有很实用的价值。

1.12　函数传值与引用

自定义函数是大多数编程语言都具备的特性，在 PHP 开发中则更具灵活性。

在 PHP 中，对函数参数个数没有限定，但是过多的参数会对调用和维护产生影响。

下面我们一起讨论函数传值的两种形式。

1.12.1　传值

调用函数多采用值传递，告诉函数去完成什么任务。函数中接收参数，只需要在函数头的括号内加入相应的变量名。格式如下：

```
$user = getUserInfo($1,$2,$3)
```

那么在函数参数定义时，我们使用如下格式来接收值的传递。

```
function getUserInfo($first,$second,$third){
    //etc
}
```

当然我们也可以使用 func_get_arg() 函数来直接处理：

```
function getUserInfo(){
    $first=func_get_arg(0);
    $second=func_get_arg(1);
    $third=func_get_arg(2);
}
```

如果你觉得还是烦琐，还可以将 func_get_arg() 函数返回的内容交给数组，然后再进行处理：

```
function getUserInfo(){
    $args=func_get_args();
    $first=$args[0];      // 数组第一个元素索引是0
    $second=$args[1];
    $third=$args[2];
}
```

在 PHP 中使用值传递为变量赋值，比如当把一个变量的值分配给另一个变量时，其实是替代另一个变量的原值。变量相当于计算机存储器的一个代号，比如：

```
$a = $b;
```

这一行代码会把 $b 变量里的值替换为 $a 的值，之后也不影响变量 $a，从优化存储数据角度来看，这并非最佳的方案。因此便有了引用的传值方法。

1.12.2　引用

在 C 或 C++ 里，我们都知道有个“指针”的概念。它是一个指向内存地址的变量，这也是被称为指针的原因。C++ 的指针在内部，对于开发者来说不可见，它的特点是可

直接访问到需要的内容，速度更快。

PHP 的指针与这些语言机制相同，即可以用一个变量名称把工作地址与原始存储位置建立一个对应关系。如果我们在一个函数的参数前加入引用，这表示当函数对该内部变量的值进行修改时，同时也能够反映到函数外部，需要在对应的参数前加上“&”符号，如代码清单 1-19 所示：

代码清单1-19　使用引用传递参数

```
function build_row(&$text){
    $text = "<tr><td>$text</td></tr>";
}
echo '<table border="1">';
$t = '测试数据';
$row = &build_row($t);    //引用方式调用函数

echo $t;
echo $t;
echo $t;
echo '</table>';
```

这个脚本将打印一个 3 行的表格，函数的功能用来打印一行，如图 1-3 所示。

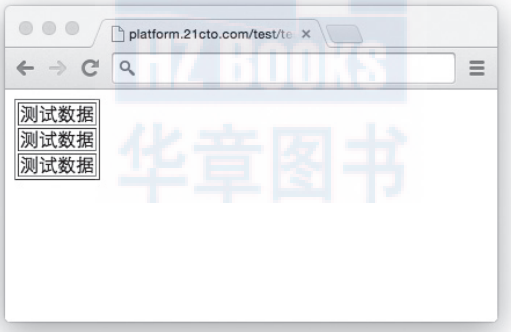

图 1-3　通过函数来打印内容

使用引用的特征就是参数有“&”符号，它确定是否采用引用传递，这个符号的存在使得它能够“感受”到函数内部对它的变更，在函数调用完成时，针对这个变量的任何修改将同步跟进。所以按值和按引用的差别也就在这里。

注意，调用函数时也需要用 & 符号来声明是引用操作。

1.13　避免使用过多参数

在开发中，我们要尽量在函数或方法中避免使用过多的参数。首先可维护性不好，其次在调用时写起来也麻烦，一不小心就可能被提示缺少参数。

因此，当参数过多、过长时，就要考虑我们的思路是否需要修正。如果参数过多的情况无法避免，可以利用全局变量，但是这种方法不提倡。

下面我们就来讨论如何避免函数参数过多的解决方案，应该有一款风格适合你。

1.13.1　使用数组

在函数中可以使用标量变量作为参数，也可以使用数组作为参数，从而有效减少函数参数的数量。

这样在调用函数时可用如下形式：

```
$bar = func(array('dude','where is my','car'));
```

函数定义如下：

```
function func($args){
    $first = $args[0]; // 数组的第一个元素从0开始
    $second = $args[1];
    $third = $args[2];
}
```

可以看到调用函数时使用数组，PHP 函数将数组元素作为单独的变量来进行处理。

1.13.2　使用对象

我们知道，对象是一个类的实例，当使用对象传递给函数或方法时，方法或函数中就可以调用对象提供的全部公有和私有方法，而不只是一个参数。

在下面实例中有一个 User 用户类和一个 UserCsvTemplate 类，有一个方法来显示一个 CSV 类与 CSV 的用户数据，如代码清单 1-20 所示：

代码清单1-20　使用对象传递参数

```
<?php
class User{
    public $user_name:
    public $type;
    public $email;
    public $address;
```

```
    public $city;
    public $country;
    public $gender;
    //...
}
class UserCsvTemplate{
    public function render($user_name, $type, $email, $address, $city, $country,
        $gender){
        echo $user_name, ';',
        $type, ';', $email, ';',
        $address, ';', $city, ';',
        $country, ';', $gender, PHP_EOL;
    }
}
?>
```

我们想调用 UserCsvTemplate 类，首先新建 User 对象，然后再将值传递。如代码清单 1-21 所示：

代码清单1-21　使用值传递参数

```
$user = new User();
...
$csv_template = new UserCsvTemplate()
$csv_template->render($user->user_name, $user->type, $user->email, $user->
    address, $user->city, $user->country, $user->gender);
```

可以看到上面的 render() 方法包含非常多的参数。

由于所有的参数属于一个对象，可以直接传递，如代码清单 1-22 所示：

代码清单1-22　使用对象传递参数

```
<?php
class UserCsvTemplate{
    public function render(User $user){
        echo $user->first_name, ';', $user->last_name, ';',
        $user->type, ';', $user->email, ';',
        $user->address, ';', $user->city, ';',
        $user->country, ';', $user->gender, PHP_EOL,
    }
}
$user = new User();
//...
$csv_template = new UserCsvTemplate();
$csv_template->render($user);
```

1.14　匿名函数

匿名函数由 PHP5.3 引入，也称为动态函数，在 PHP 5.4 后有了进一步扩展。下面是一个简单的匿名函数的例子。如代码清单 1-23 所示：

代码清单1-23　使用匿名函数

```
<?php
$greet = function($name){
    printf("Hello %s\r\n", $name);
};
?>
```

初看上去很奇怪，其实仔细看与赋值操作很像——如同将一个变量赋值为字符串、整数一样，只不过这次是给一个函数赋值，也就是在后面以分号结束。

从代码中看到，我们调用这个函数直接使用该变量名字增加括号就可以了。由于该函数有一个参数 $name，我们将它放在括号里就可以了。

```
 $greet('World');
 $greet('PHP');
```

有一个更简单使用匿名函数的方式。PHP 中的 array_map() 函数返回用户自定义函数作用后的数组。回调函数接受的参数数目应该和传递给 array_map() 函数的数组数目一致。也就是说，array_map() 函数接收一个函数作为其第一个参数，第二个参数是数组，数组内的每个元素都将使用之前的函数遍历一遍。

```
function format_names($value){
    //etc
}
array_map('format_names' , $names);
```

上面的代码中，严格来说函数是有名字的。我们再换用匿名函数的方式来处理，格式如下：

```
array_map(function($value)){
    //etc
},$names);
```

这种方式的好处是：相关代码、函数定义与隐式调用结合更紧密，因为直接使用函数，只需要维护匿名函数定义即可。

使用匿名函数的副作用是，有可能出现解释出错。倘若发现这样的错误，可以把函数中的代码放在一个正常的函数体中执行，调试到没有问题为止。

匿名函数可以使用闭包。这种方式在 PHP 中比较少用，但在 JavaScript 中会常用到。如果你的 PHP 版本小于 5.3，需要使用如下方式：

```
$foo = create_function('$x', 'return $x*$x;');
$bar = create_function("\$x", "return \$x*\$x;");
echo $foo(10);
```

PHP 5.3 以前版本不支持闭包，变量需要显式声明。代码样式如下：

```
$x = 3;
$func = create_function($z) { return $z *= 2; };
echo $func($x); // 打印结果为 6
```

在 PHP 5.3 以后，我们就可以使用以下方式了：

```
$x = 3;
$func = function() use(&$x) { $x *= 2; };
$func();
echo $x; // 打印结果为 6
```

闭包语法要优于 create_function()，当我们使用 create_function() 时，参数和函数体都要显式声明，也就是说在代码运行前，PHP 无法解析其语法正确性，需要特别注意单引号、双引号和变量命名规则。

当使用闭包后，PHP 可以像检查正常代码一样，对匿名函数进行检查。

1.15 return 与 exit

我们知道，return 函数是专门用在函数体和方法里的，在调用函数和方法时，使用 return 从函数内部返回调用处。

exit 语句的功能是在代码逻辑里中断执行，因此 exit 语句在函数体中少用或不用。在一些框架的方法中，如 CodeIgniter、Laravel、YII，甚至 ThinkPHP，想要在方法里中断函数的运行并返回调用处时要使用无返回值的 return，而不要使用 exit 或 exit() 函数。

1.16 is_callable() 与 method_exists() 函数

在很多产品应用中，我们经常能够看到以下这种用法，它用来检查一个对象里的方法是否存在。如代码清单 1-24 所示：

代码清单1-24　检查一个对象中的方法是否存在

```
<?php
if (method_exists($object, 'SomeMethod')) {
    $object->SomeMethod($this, TRUE);
}
?>
```

这段代码的目的比较容易理解，有一个对象为 $object，我们想知道它是否有一个方法为 SomeMethod，如果有，就调用此方法。

这个代码看起来正确，而且在大部分的时候运行也会正常。但是如果这个 $object 对象的方法对于当前的运行环境是不可见的，程序还能正常运行吗？正如这个函数名方法存在一样，只是对我们提供的类或对象检查是否有我们所期望的方法，如果有，就返回 TRUE，如果没有，就返回 FALSE，这里并没有考虑可见性的问题。所以，当你恰好判断一个私有或者受保护的方法时，你能够得到一个正确的返回，但是执行的时候，会得到一个“Fatal Error”错误警告。

上面这段代码的真正意图应该理解为：对于提供的类或者对象，我们能否在当前的作用域中调用它的 SomeMethod 方法。而这正是 is_callable() 函数存在的目的。

is_callable() 函数接收一个回调参数，可以指定一个函数名称或者一个包含方法名和对象的数组，如果在当前作用域中可以执行，就返回 TRUE。如代码清单 1-25 所示：

代码清单1-25　使用is_callable()函数

```
<?php
if (is_callable(array($object, 'SomeMethod'))) {
   $object->SomeMethod($this, TRUE);
}
?>
```

下面我们举一个例子，来说明 method_exists() 和 is_callable() 函数的区别。如代码清单 1-26 所示：

代码清单1-26　method_exits()和is_callable()函数的区别

```
<?php
class Foo {
    public function PublicMethod(){}
    private function PrivateMethod(){}
    public static function PublicStaticMethod(){}
    private static function PrivateStaticMethod(){}
}
```

```
$foo = new Foo();

$callbacks = array(
    array($foo, 'PublicMethod'),
    array($foo, 'PrivateMethod'),
    array($foo, 'PublicStaticMethod'),
    array($foo, 'PrivateStaticMethod'),
    array('Foo', 'PublicMethod'),
    array('Foo', 'PrivateMethod'),
    array('Foo', 'PublicStaticMethod'),
    array('Foo', 'PrivateStaticMethod'),
);

foreach ($callbacks as $callback){
    var_dump($callback);
    var_dump(method_exists($callback[0], $callback[1]));
    var_dump(is_callable($callback));
    echostr_repeat('-', 40);
    echo '<br />';
}
?>
```

执行上面的脚本后，我们会清晰地看到两个函数间的差别。

is_callable() 还有其他的用法，例如，不检查所提供的类或方法，只检查函数或方法的语法是否正确。像 method_exists() 一样，is_callable() 可以触发类的自动加载。

如果一个对象存在魔术方法 _call，在进行方法判断时 method_exists() 会返回 FALSE，而 is_callable() 会返回 TRUE。如代码清单 1-27 所示：

代码清单1-27　使用__call魔术方法

```
class MethodTest {
     public function _call($name, $arguments){
   echo 'Calling object method ' . $name . ' ' .implode(', ', $arguments);
   echo '<br />';
    }
}

$obj = new MethodTest();
$obj->runtest('in object context');
var_dump(method_exists($obj,'runtest'));
var_dump(is_callable(array($obj,'runtest')));
echo '<br />';
```

下面我们总结一个表格，对比两个函数的详细区别（见表 1-3 ）。

表 1-3　method_exists() 与 is_callable() 函数的比较

比较内容	method_exists()	is_callable()
调用形式	bool method_exists (mixed $object, string $method_name)	bool is_callable (callback $name [, bool$syntax_only = false [, string& $callable_name]])
适用范围	仅适用于判断类方法	可以判断全局函数，也可以判断类方法
是否有上下文	否	是，会判断一个函数是否在当前环境中可调用（例如在子类中判断能否调用父类构造函数）
是否判断权限	否	是，在类外，判断 private 和 protected 方法会返回 FALSE
是否调用 _call 方法	否	是
速度	快	慢

1.17　执行外部程序

在 PHP 中用 ` 来运行外部系统命令或应用程序。你可能也看得出来，这不是单引号，而是键盘左上角 ESC 键下方的上档键，它被称为反引号操作符（backtick）。我们使用如下代码：

```
$out = `dir c:`;            //适用于Windows或部分Linux系统
echo $out;
```

这段代码调用了 Windows 系统命令 dir 显示 C 盘根目录下子目录和文件信息。下面稍改动一下在 Linux 系统的外部命令运行目录列表：

```
$out = `ls -al`;            //适用于Linux系统
echo $out;
```

我们还可使用另外一个函数 shell_exec() 来执行外部程序或命令：

```
$out = shell_exec("dir");
echo $out;
```

两个函数得到的结果相同，稍有所区别的是，用 ` 符号会将返回结果放在一个数组，而 shell_exec() 函数则是将返回结果放在一个标量变量中。

1.18 安全模式的使用说明

PHP 安全模式用来限制用户使用外部命令或执行不安全的操作，这种配置多用在保密级别高的服务器、云主机或虚拟主机等多用户环境中。

打开或者关闭安全模式，需打开 php.ini 中的 safe_mode 选项即可：

```
safe_mode  =  on
```

修改完成，重启 Apache 或 Nginx 后即可生效。这样在执行或试图要打开外部文件时，PHP 会审核所有者和其本身所有者是否匹配，防止越权执行或修改。

启用安全模式会对 PHP 本身的行为产生影响，限制函数的功能以及禁用部分函数。以下函数会受到影响：

chdir, move_uploaded_file, chgrp, parse_ini_file, chown, rmdir, copy, rename, fopen, require, highlight_file, show_source, include, symlink, link, touch, mkdir, unlink, putenv, set_time_limit, set_include_path。

这些函数大多数与文件、目录及系统相关，此外 dl() 这个动态加载扩展库函数功能也会被禁止执行。如果要加载外部扩展库，需要在 php.ini 文件中添加相应库，使其在运行环境时加载进来。

当安全模式开启，要在 PHP 脚本中执行外部程序时，需在 php.ini 中的 safe_mode_exec_dir 选项指定外部程序的所在目录，否则会无法执行，同时也会自动传递给 escapeshellcmd() 函数进行过滤。

一些外部命令执行函数也会受影响，包括 exec, shell_exec, passthru, system, popen，以及外部命令执行操作符 (``)。

当脚本在访问文件系统时会进行所有者检查。默认情况会检查该文件所有者的用户组 id，用户组 id(gid) 在 safe_mode_gid 选项中指定。

若要在安全模式下脚本中使用 include 或 require 包含，需用 safe_mode_include_dir 选项来设置包含文件所在的路径，以保证代码正常工作。例如，包含 /usr/local/include/php 下的文件，可以设置选项为：

```
safe_mode_include_dir = /usr/local/include/php
```

如果在脚本中运行外部命令，需要修改 php.ini 里的 safe_mode_exec_dir 参数，例如想要执行 /usr/local/php-bin 路径下的文件，可以改成：

```
safe_mode_exec_dir = /usr/local/php-bin
```

如果想修改某些系统环境变量，可使用 safe_mode_allowed_env_vars 选项。这个选项的值是一个环境变量的前缀，默认允许 php_ 开头的环境变量，如果想要修改，可以设置该选项的值，多个环境变量之间使用逗号分隔。

1.19　提前计算循环长度

1. 优化前后的对比

在进入一个循环之前计算长度是另一项可以优化的技术。以下代码是一个简单的 for 循环，它将遍历数组 $item 并分 10 次计算出数值，以确定哪些地方可以进行优化。代码清单 1-28 如下所示：

代码清单1-28　未优化的for循环

```
<?php
$items = array(1,2,3,4,5,6,7,8,9,10);
for($i=10;$i<count($items);$i++){
    $x = 1999 * $i;
}
?>
```

我们主要注意 for 循环的逻辑，PHP 按以下方式来执行此循环。

（1）初始化 $i 变量为 0，从索引 0 开始循环，使用 count() 函数计算数组长度，$i 计数器加 1。

（2）迭代 0 完成后，开始索引 1，使用 count() 计算数组长度，$i 计数器加 1。

（3）迭代 1 完成后，开始索引 2，使用 count() 计算数组长度，$i 计数器加 1。

代码继续运行，直到到达数组元素的末尾。这段代码在开始时，问题便存在了。每次开始一个新索引的循环时，都必须调用函数 count() 来确定数组长度。上面的代码中，count() 被调用了 10 次，其中有 9 次是多余的，因此这些不必要的调用需要替换，只要到达 for 循环之前调用一次 count() 就可以了。

那么修正优化后的 for 循环如代码清单 1-29 所示：

代码清单1-29　优化后的for循环

```
<?php
$items = array(1,2,3,4,5,6,7,8,9,10);
$count = count($items);
for($i=10;$i<$count;$i++){
```

```
    $x = 1999 * $i;
}
?>
```

上面的代码产生的结果与前一个代码清单的执行结果相同，但是函数的调用次数却从 10 次降到了 1 次，很明显的道理，调用的次数越少，PHP 执行的程度就越快。

2. 计算优化节省的时间

为了准确计算出减少 9 次 count() 函数调用能够节省多少时间，我们可以使用 microtime()。在代码清单 1-30 中，增加另一个 for 循环，执行该代码 10 万次，代表有 10 万个用户来请求该脚本。我们对有变动的代码以粗体表示出来。

代码清单1-30　未优化的for循环基准代码

```
<?php
$items = array(1,2,3,4,5,6,7,8,9,10);
$start = microtime();
for($x = 0; $x<100000; $x++) {
    for($i=10;$i<count($items);$i++){
        $x = 1999 * $i;
    }
}
echo microtime()-$start;
?>
```

执行 10 次该代码并计算结果的平均值，我们得出，10 万次循环的总执行时间为 0.046 ms。重新启动 Web 服务器，现在我们测试上面优化后的代码，如代码清单 1-31 所示：

代码清单1-31　优化后的for循环基准代码

```
<?php
$items = array(1,2,3,4,5,6,7,8,9,10);
$count = count($items);
$start = microtime();
for($x = 0; $x<100000; $x++) {
    for($i=10;$i<$count;$i++){
        $x = 1999 * $i;
    }
}
echo microtime()-$start;
?>
```

我们再次运行 10 次代码并获取平均值，使用此代码我们会看到，for 循环的平均执行时间为 0.0095 ms，减少了 0.036 ms，即优化过的代码快了 0.036 ms。

3. 使用 foreach 替代 for 和 while 循环

访问数组数据的方法也是可以优化的，那就是尽量使用 foreach 语句，让它来替代 while 和 for 循环。

优化数据访问的方法对性能来讲很重要。许多 Web 应用需要从数据库、XML、JSON 文件中读取数据并必须遍历每条记录后才能将数据显示给用户，能减少一毫秒等待，对于产品和用户来说，都有很重要的价值。

还是使用实例来说明这种优化之细节，如代码清单 1-32 所示：

代码清单1-32　使用foreach语句

```
<?php
$items = array_fill(0,100000,' 12345678910' );
$start = microtime();
reset($items);
foreach($items as $item)
{
    $x = $item;
}
echo microtime()-$start;
?>
```

该脚本创建了一个数组 $items，其中包含 10 万个元素，每个元素含有 155 个字节的字符串，代表数据库中典型数据。之后该代码设置了开始时间并使用 foreach 循环访问数组的每个元素，最后我们以毫秒为单位显示所用时间。我们连续执行了 10 次上面的代码清单，然后计算出每次执行时间的平均值，其结果为 0.0078ms。

我们以上面的代码为基础，使用 while 循环，而不是 foreach 循环。代码清单粗体部分为我们对其做的修改，如代码清单 1-33 所示：

代码清单1-33　使用while循环

```
<?php
$items = array_fill(0,100000,' 12345678910' );
$start = microtime();
reset($items);
$i=0;
while($i<100000){
    $x = $items[$i];
    $i++;
}
echo microtime()-$start;
?>
```

重新启动 Web 服务器，运行该代码 10 次以后，我们再次计算平均执行时间。用 while 循环访问数组中单个元素的平均时间为 0.0099 ms。

下面我们再来比较一下使用 for 循环，如代码清单 1-34 所示。我们按照同样的基准循环流程，重启 Web 服务器，执行代码 10 次并计算平均结果。

代码清单1-34　使用for循环

```
<?php
$items = array_fill(0,100000,' 12345678910' );
$start = microtime();
reset($items);
$i=0;
for($i=0;$i<100000;$i++){
    $j = $items[$i];
}
echo microtime()-$start;
?>
```

我们将上述 3 种循环基准的结果总结在表 1-4 中。

表 1-4　10 个元素数组的 PHP 循环平均执行时间

循 环 类 型	平均执行时间（ms）
foreach	0.0078
while	0.0099
for	0.0105

由此可见，使用 foreach 循环是访问数组元素性能最优之方法。感兴趣的朋友也可以自己来尝试一下。

1.20　SQL 组合优化

在开发过程中，有一些代码在语法上没有任何问题，但在执行上会有较大的时间成本浪费。比如有这样的代码，如代码清单 1-35 所示：

代码清单1-35　未经优化的SQL执行

```
<?php
$user_ids = array(101200,101201,101202,101203);
foreach($user_ids as $user_id){

    $sql = "SELECT * FROM users WHERE user_id= $user_id ";
```

```
    $res = mysql_query($sql);

    //Execute Query
    ......

}
?>
```

大家看到这种代码，是不是有种似曾相识的感觉？是的，这种代码对一些开发者来说并不陌生。这种程序每次都要重复遍历查询 MySQL，造成时间的浪费和数据库的压力。

我们不妨转换成下面的代码样式，如代码清单 1-36 所示：

代码清单1-36　优化后的SQL查询

```
<?php
$user_ids = array(101200,101201,101202,101203);

$user_ids_str = "'".implode("','",user_ids)."'";
$sql = "SELECT * FROM users WHERE user_id= $user_id ";
$res = mysql_query($sql);

//Execute Query
......
?>
```

从源代码上来做比较，第二段代码只执行了一次查询就已经完成了任务。

类似于这样的代码很多。它们从语法上并无显式错误，但性能上可差了不少。因此，在开发中，需要我们更多地深入考虑细节，机器是按我们的“旨意”在执行的，效率和成本取决于人。

1.21　文件处理

文件系统处理包括文件和目录的新建、复制、移动、删除等操作。在开发前我们要确认脚本对某个文件和目录有相应的文件系统的读写权限，包括安全模式及 Apache 或 Nginx 的权限设置，以保证 PHP 对文件的操作正确性。

fopen() 函数会把操作的文件句柄放在文件头，如代码清单 1-37 所示：

代码清单1-37　文件打开的几种方式

```
//打开Linux系统下的文件
$handle = fopen("/var/logs/somefile.txt", "r");
```

```
// 打开Windows系统下的文件
$handle = fopen("c:/data/info.txt", "rt");
$handle = fopen("c:/data/info.txt", "rt");
```

PHP 能提供几种打开文件的方法和模式，具体取决于我们想用它做什么。如果处理已经存在的文件，也不想删除它原来的内容，可用下面两种模式。

- R-：以只读方式打开该文件，将游标置于文件头。
- R+-：打开文件进行读取和写入，将游标置于文件头。

如果想新创建一个文件或替换现有文件，可使用以下两种模式之一。参数说明如下：

- W- ：打开文件，将光标置于文件头，如果该文件存在，将清空它的内容（零长度截断该文件），如文件不存在，它将尝试创建。
- W+-：同上，这一次的打开文件进行读也一样。

PHP 允许我们使用两种追加写入文件的模式。参数说明如下：

- a- ：追加模式。打开文件，如果文件有内容，则从末尾追加写（读）。如果该文件不存在，则尝试创建它。
- a+- ：追加模式。打开文件，游标置于文件尾部，将从文件末尾开始追加（写），如果该文件不存在，则尝试创建它。

以下两个模式，称为谨慎的文件写操作：

- X- ：写模式打开文件。从文件头开始写，如果文件已经存在，该文件将不会被打开，fopen() 将返回 FALSE，PHP 也会产生警告。
- X+-：读 / 写模式打开文件。功能和 X- 相同。

当基于 Windows 的操作系统上的文件处理时有两个模式，开发者需要了解一些参数的处理。

- t：当前处理的文本文件的行结束符（\r\n）。
- b ：如果处理的是非文本文件，建议使用 b 标志；如果不这样，在 Windows 系统打开文件时，可能会遇到一些奇怪的问题，因此开始需要使用此模式。

下面我们将专注说明读取文本文件的几个方法。

大多的读取文件场景，都是读取文件的每一行后进行操作。这时可以使用 file() 函数读取整个文件到一个数组中，如代码清单 1-38 所示：

代码清单1-38　使用file函数打开文件

```
$lines = file("/tmp/files/InputTextFile.txt");
foreach ($lines as $line_num => $line) {
```

```
    echo "Line #{$line_num} : " . $line . "n";
}
```

我们还可以添加 file() 函数支持的可选参数：

- FILE_USE_INCLUDE_PATH - 在 include_path 包含路径中查找相关文件。
- FILE_IGNORE_NEW_LINES - 不能在每个数组元素的最后添加新行（在这里是 $line)。
- FILE_SKIP_EMPTY_LINES- 跳过空行，这在删除文件中多余空行时很有用。

file_get_contents() 函数可以一次读取文件的全部内容，它接受两个额外参数时很有用，分别是 offset 和 MAXLEN（PHP 5.1 以上支持），offset 偏移指定从哪里开始读取，MAXLEN 指定从源文件读取的字节数。

下面的代码表示从第 128 字节开始读取，读取 1KB 的数据内容，如代码清单 1-39 所示：

代码清单1-39　分段读取文件

```
$file = file_get_contents("/tmp/files/InputTextFile.txt",0,null,128,1024);
echo $file;
```

可以使用 file_get_contents() 函数读取远端 URL 的文件内容，如代码清单 1-40 所示：

代码清单1-40　读取远端URL文件

```
$file= file_get_contents("http://www.21cto.com/files/InputTextFile.txt");
echo $file;
```

还可以联合使用 file_get_contents() 和 file_put_contents() 函数，它接受一个连续的内容作为参数，并一次写入文件里，如代码清单 1-41 所示：

代码清单1-41　读取文件和写入文件

```
//某图片的url地址
$url="http://www.21cto.com/assets/images/logo.png";
//读取二进制“字符串”
$data=file_get_contents($url);
//要写入的目标文件和路径
$filepath = "/usr/local/www/images/upload/upload.jpg";
//保存
file_put_contents($filepath,$data)or die("不能写入文件");
```

我们可以使用这两个函数很方便地抓取远端文件，如 HTML 页面或图片等资源。还有更直接的方式来取得缓冲区中的文件内容，使用 readfile() 函数可以做到这一点。我们

甚至不需要做更多的处理，它已返回已读取的字节数，如代码清单 1-42 所示：

代码清单1-42　使用readfile()函数获取文件的大小

```
<?php
// 使用readfile()
$file = "/tmp/files/InputTextFile.txt";
$bytesRead = readfile($file);
echo $bytesRead;
?>
```

fgets() 函数可以帮我们读取文件时从文件指针的位置开始读取一行，并作为一个字符串返回，也可以指定想让它读取的字节长度。在下面的例子中，我们想读取文件中的 8KB 字节。具体如代码清单 1-43 所示：

代码清单1-43　指定读取文件的长度

```
<?php
$file = "c:/tmp/files/InputTextFile.txt";
$handle = fopen($file, "rt");
if ($handle) {
    while (!feof($handle)) {
        $buffer = fgets($handle, 8192);
        echo $buffer;
    }
    fclose($handle);
}
?>
```

最后一个函数，我们看一下如何使用 fread() 函数，它主要用来读取二进制文件。它需要一个文件句柄和文件指针读取字节的长度。读取文件结束时，当文件或网络数据包（流）被读取到 8192 个字节（ 8 KB），或者已经到文件尾（ EOF）时，读取结果。具体如代码清单 1-44 所示：

代码清单1-44　fread()——安全读取二进制文件

```
<?php
$file = "/tmp/files/picture.gif";
// 如果是Windows系统用"rb"
$handle = fopen($file, "r");
$contents = fread($handle, filesize($file));
fclose($handle);
?>
```

下面的代码是如何从一个网址读取一个二进制文件。如代码清单 1-45 所示：

代码清单1-45　安全读取远端二进制文件

```
<?php
$handle = fopen("http://www.21cto.com/picture.gif", "r");
$contents = '';
while (!feof($handle)) {
    $contents = $contents.fread($handle, 8192);
}
fclose($handle);
?>
```

1.22　goto 语句：最后的手段

PHP5.3 后推出了极富争议的局部 goto 语句（局部是指它不可能跳出例程或者进入循环）。对一些语言，特别是 C 语言，可以执行非局部跳转，或者长跳，但是 PHP 暂不支持此特性。对于局部 goto 语句的限制，PHP 与其他语言相同，即不跳入循环体且不跳出当前的子例程。

goto 语句是作为编程中没有办法的最后手段，一般情况不经常使用，请各位尽量不要使用。goto 的语法很简单，如代码清单 1-46 所示：

代码清单1-46　使用goto语句的脚本

```
<?php
for($i=0,$j=50; $i<100; $i++) {
    while($j--) {
        if($j==17) goto end;
    }
}
echo "i = $i";
end:
echo 'j 此时等于 17';
?>
```

标签为冒号结束。这段代码的结果为：

```
J此时等于17
```

虽然 goto 语句通常的用法让人诟病不已，有一幅漫画描述了一个程序员用户画了一只恐龙，因为使用 goto 过多，结果恐龙跳出来咬这个程序员。更有人将此语句引申到一种 eval（罪恶）。

然而我认为，goto 既然作为一种选择，并不是一件坏事，它的目的也是给我们提

供简单、清晰、高效的代码，如果 goto 能够达到这些目的，那么它的存在便是有意义的。

1.23 利用 phar 扩展来节省空间

在 Java 中有 *.jar（Java archive）文档，它的本质是能将多个文件压缩到单个文件，类似于 rar 或 zip 文件包，但是 jar 或 war 可以作为应用来执行。

在 PHP5.3 以后，PHP 的 phar 扩展也可以实现 Java 这样的档案功能。它允许开发者创建或操作 PHP 档案文件，也就是名称的由来——PHP archive。

在下面的代码里，它包含了两个文件：wild.php 和 domestic.php。为了分发应用，需要分发 3 个文件。如果有更多的类，要分发文件的数量更多。只分发这两个文件的目的是：自身执行脚本，且 phar 文件包含了所有必要的类文件，如代码清单 1-47 所示：

代码清单1-47 使用phar引用文件

```
<?php
include('phar://animals.phar/wild.php');
include('phar://animals.phar/domestic.php');
$test = animal();
printf("%s",$test->get_type());
$test1= new \wild\animal();
printf("%s",$test1->get_type());
?>
```

上面代码的诀窍在于 include 指令，它引入了 animals.phar 文件并全部引用这些文件。

那么，我们讨论一下如何创建类文件？正如 Java 提供 jar 外部程序文件一样，PHP5.3 也提供了称为 phar 的外部应用程序。

创建一个 phar 文件很简单，语法如下：

```
phar pack -f animals.phar -c gzwild.phpdomestic.php
```

pack 参数指明了 phar 程序用来创建以 -f 选项指定的文件名的压缩档案包，并加入 wild.php 和 domestic.php 两个文件到压缩包中。为了能够成功运行，php.ini 配置文件中的 phar.readonly 参数需为 off，如果默认值为 on，会阻止创建新档案。我们使用的压缩算法为 zip，phar 支持的压缩算法包括 zip、gz（gzip）和 bz2（bzip2）。

phar 改变了 PHP 应用分发和打包的方式，并节省了存储空间。虽然不像命名空间或

Nowdoc 等特性那样吸引人注意，但对 PHP 应用分发方式开始有影响，比如一些主流开源程序如 phpMyAdmin、WordPress 等，都已经或开始尝试以 phar 方式发布。

与 Java 的 jar 包一样，亦无须担心性能问题，phar 包只被解析一次。在脚本开始时间占得非常小，不影响执行时间。

1.24　手册上的小瑕疵

到这里，我在想很多人，也包括我自己，以前会写类似这样的代码：

```
echo "欢迎".getUserInfo()."，购物车商品数量：".ShowShopCart();
```

乍一看代码写得没有什么毛病，这种相似的代码例子在 PHP 手册上就存在。从编译上来看，这段字符串在输出之前实际上运行了 3 次！这是为什么？我们看一下 PHP 是如何执行这段代码的。

PHP 解析与执行这段代码的步骤，是这样的：

（1）创建一个新的临时字符串。

（2）把“欢迎”加入字符串。

（3）把调用 getUserInfo() 函数返回的内容加入字符串。

（4）创建一个新的临时字符串。

（5）放入第一次创建的字符串。

（6）把“购物车商品数量”加入字符串。

（7）把调用 ShowShopCart() 函数返回的内容加入字符串。

（8）发送最终的临时字符串，打印在屏幕上。

怎么样？看起来很简单的功能，现在似乎有点儿复杂。解决的方法是，使用 echo() 函数的另一种写法，把小圆点换成逗号，除了具有原来相同的功能，不再新建字符串，而只是一个字符串连接操作，然后直接输出。

```
echo "欢迎",getUserInfo(),"，购物车商品数量：",ShowShopCart();
```

请看，就这么简单，一个逗号完美地解决了性能问题。

在如今的 Web 应用中，大量使用了缓存技术，它可以降低编码侧的性能问题。但还是需要开发者在写程序时不要偷懒，多留心、留意——其实程序员与工程师的区别也在这里。

1.25 本章小结

在本章，我们一起深入解析了在日常开发中易出现模糊的技术点，更多的是在开发场景中会用到的实用内容，另外也包括一些开发中的高级的以及实用的知识。

正如你看到的，一些看似简单的代码却时常让人迷惑。比如在细节上，在编程思路上，陷入逻辑误区。我们列出一些常见的问题，并给出了解决方案。

另外，本章也对 PHP 新旧版本功能进行了深入理解与区分。祝你编码愉快。

第 2 章 Chapter 2

深入 PHP 面向对象

面向对象开发（Object Oriented Programming，OOP）。在 PHP 中也被大部分的互联网开发团队采用。

PHP 面向对象开发的发展历程与软件工程的发展极其相似，都是从过程化到模块化，到面向对象。幸运的是，面向过程、面向对象两种模式在 PHP 里都能很好被支持。

在一个项目刚开始时，可能会以使用面向过程为主，而部分使用面向对象的开发形式，这样也具有不错的灵活性。当项目发展到一定阶段时，比如团队技术层面整合、扩展性不佳、维护变困难等问题便凸显出来了。其实使用面向过程开发也没什么不好，如果我们能遵守好既有的规则，比如代码和目录结构，开发效率和可维护也可以兼顾，这在我曾经的项目中也应用过，至今可维护性也不错，面向过程开发的代码在性能上要优于面向对象。

但是大多数的状况是，产品要快上线，日积月累，不同的人重复造轮子，代码质量参差不齐，胶水式的代码遍布 SVN，上面提到的各种维护性问题开始出现。全面使用面向对象编程可以改变这些现状，它可实现的目标如下：

- 容易在已有代码的基础上扩展。
- 允许类型微调，以在方法中对这些变量进行权限控制。
- 结合设计模式，能够解决大多数软件设计的问题，扩展性好，调试更容易。

虽然会稍稍损耗一些性能，但面向对象开发的价值远远大于性能。它的重要性在于

封装，这也是它在 PHP 项目中使用越来越多的原因。

PHP 面向对象开发与 Java、Microsoft .NET 等概念相通，但一些细枝末节也有少许不同，需要留意。

在本章中，我将和大家一起深入讨论 PHP 面向对象。主要的主题如下：

- 面向对象基本知识。
- 实例与多态。
- 抽象类与接口。
- 面向对象开发实例。
- 面向对象开发和性能调优。

2.1 PHP 与面向对象

从过往经历来说，面向对象是从初学者通向开发者的一把钥匙，是个陌生又奇妙的内容。我们要学会使用正确的思想来设计，将业务逻辑、过程化抽象为面向对象思维，重要的是面向对象开发实践。

因此，使用面向对象开发可以帮助我们解决下列问题：

- 方便扩展：代码重构和重用。
- 允许方法和成员变量隐藏，可控制哪些变量不允许被访问。
- 使用设计模式，解决常见软件设计问题。
- 让代码调试更容易，可维护，效率更高。

面向对象编程的目的是让开发者的生活轻松。它可将问题分割成小块，容易理清并易解决。

当然，它最重要的好处就是节约我们的开发时间！

2.2 面向对象的一些概念

在开始之前，先介绍一些面向对象开发的概念，略枯燥，但重要。

对象：对象是一些变量和方法都聚集在一起的实体。开发中可调用对象实例，而不是调用变量或方法。在一个对象中有属性和方法，功能在方法中，属性是对象的信息。

类：类相当于蓝图，它可以称为对象的模板。它描述如何创建一组代码，定义如何将一个对象的行为和相互影响或者如何使用它。当每次创建一个 PHP 对象时，实际上是

调用了类。所以有时我会在文中将类和对象放在一起说，因为它们都是同义词。

属性：一个属性是类中一个内部变量，可以保留一些信息的容器。不像其他的语言，PHP 不会检查属性变量的类型。属性可以在类本身、子类，或调用之处。从本质上讲，属性是指在类本身，而不是在这个类中的任何方法中声明的变量。

方法：方法是在一个类中的函数。和属性类似，方法也按访问等级分为 3 个类型。

封装：封装是面向对象结合在一起的机制、代码和操纵数据，并防止外界干扰和误用。包装好的数据和方法到一个类作为封装。封装之后，我们不用担心执行的任务没在里面。

多态性：对象可以是任何类型。一个离散的对象可以有属性和方法，离散的工作分别到其他对象。一组对象可以来自父类或保留父类的一部分属性，这个过程被称为多态性。一个对象可以演变为保留其行为的其他派生对象。

传承：通过扩展派生一个对象成为一个新对象的过程称为继承。当你从另一个对象继承时，子类（即继承）派生的所有属性和方法的超类（这是继承）。子类可以沿用各种父类方法。

耦合：耦合是类如何彼此依赖的行为。松耦合比紧耦合的对象更容易重用。在下一章我们将了解耦合的细节。耦合是面向对象开发中很重要的问题。

设计模式：面向对象编程的技巧，用更聪明的办法解决类似的问题集合。使用设计模式（Design Patten）可以让你用最少的代码开发最好性能的应用程序。有时设计不佳导致问题，可以使用设计模式解决方案来优化。但是，不必要和无计划的使用设计模式也会降低应用性能。

子类：一个面向对象开发中很常见的名词，在本书中我们使用这个词。当一个对象从另一个对象派生，派生一个新类，这称为子类。

父类：如果对象是从它派生的，这个类就称为超类或父类。为了保持简单，当你扩展一个对象，则该对象是一个新扩展对象的父类。

实例：当你创建一个对象时，通过调用其构造函数，它会被称为一个实例。只要写如 $v= new $object 的表达式，就能创建类的实例对象。

2.3　类和对象

什么是对象？我们还可以这样理解，它是一些代码的属性和方法的堆砌。对象类似

于一个数组，数组可以存储属性（数组被称为键（key）），对象比数组多出一些方法，它们可以隐藏或开放存取，这在数组里是做不到的。

对象是一个数据结构，可以建立一个松散耦合或多个紧耦合。我们来看一段 PHP 面向对象的代码。这是一个简单的类，功能是发邮件给用户，如代码清单 2-1 所示：

代码清单2-1　一个邮件发送类

```
<?php
class emailer{
    private $sender;
    private $recipients;
    private $subject;
    private $body;
    function __construct($sender){
            $this->sender = $sender;
            $this->recipients = array();
    }
    public function addRecipients($recipient){
            array_push($this->recipients, $recipient);
    }

    public function setSubject($subject){
            $this->subject = $subject;
    }
    public function setBody($body){
            $this->body = $body;
    }
    public function sendEmail(){
            foreach ($this->recipients as $recipient){
                    $result = mail($recipient, $this->subject, $this->body,"From:
                        {$this->sender}\r\n");
                    if ($result) echo "邮件已经成功发送到    {$recipient}<br/>";
            }
    }
}
?>
```

上面的类中包含 4 个私有属性和 4 个公共方法，这些方法主要都用来处理邮件的收件人。我们现在来使用这个类，如代码清单 2-2 所示：

代码清单2-2　创建emailer类的实例对象

```
include('class.emailer.php');
$emailer = new emailer("jobs@21cto.com"); //创建新对象
```

```
$emailer->addRecipients("job@21cto.com"); //访问方法
//给一些方法发送参数
$emailer->setSubject("我要找工作");
$emailer->setBody("你好，老弟，你还好吗?");
$emailer->sendEmail();
```

可以看到，上面的代码片段的可读性都比面向过程方式清晰，从而使代码易于管理维护。知名的 CMS 与博客软件 WordPress 的开发者在他的网站上写的座右铭：“编码如诗”，你我都是作诗的人，不是么？

2.4　使用对象

在上面的代码中，我们首先创建了一个 emailer 类的实例。有一点需要大家注意，这个类需要提供一个发件人邮件地址。类似于下面这个样子：

```
$emailer = new emailer("webmaster@21cto.com"); //创建新对象实例，设置发件人
```

你应该还记得在类中的构造方法是 function __construct($sender)。

当启动一个对象时，构造方法会被自动调用，所以我们创建 emailer 类时需要给构造方法提供的正确参数。下面这样的代码将引发一个警告错误：

```
$emailer = new emailer();
```

执行上面的代码后，PHP 会提示如下警告信息，并停止执行：

```
Warning: Missing argument 1 for emailer::__construct(),
called in C:\OOP with PHP5\Codes\ch1\class.emailer.php on line 42
and defined in <b>C:\OOP with PHP5\Codes\ch1\class.emailer.php</b>
on line <b>9</b><br />
```

因此，如果类的构造方法有参数，但没有传递给它，就会触发类似上面的错误，需要各位开发时多留意。

2.5　构造方法与析构方法

构造方法是在创建对象实例时自动执行的方法。在 PHP 中，有两种方法你可以写一个内部类的构造方法。一种是创建一个名为 __construct() 的方法，也可以创建和类相同名字的方法（这个特性是为了和 PHP 低版本兼容）。

我们来看一个类代码。这个类的功能是输入任何数值后计算阶乘，如代码清单 2-3 所示：

代码清单2-3 一个阶乘计算类

```
<?php
class factorial{
    private $result = 1; //可以在外部初始化此值
    private $number;
    function __construct($number){
            $this->number = $number;
            for($i=2; $i<=$number; $i++){
                    $this->result *= $i;
            }
    }
    //显示阶乘结果
    public function showResult(){
            echo "Factorial of {$this->number} is {$this->result}. ";
    }
}
?>
```

在上面的代码中，我们使用 __construct() 作为构造函数名称。如果使用 factorial 名称的方法同时存在，它将会被重命名。

你可能会问，为何一个类中可以有两种风格的构造方法？这意味着一个类中可以有两种构造方法：function __construct() 和与类名相同的方法。

那么，PHP 会执行构造方法，还是同名方法，还是它们会一起执行？

这是一个很好的问题。其实没有执行两个方法的机会，如果是两个风格的构造方法都在类中出现，PHP 虚拟机会优先执行 __construct() 的方法，其他方法会被忽略。如代码清单 2-4 所示：

代码清单2-4 使用构造方法

```
<?php
//class.factorial2.php
class factorial{
    private $result = 1;
    private $number;
    function __construct($number){
            $this->number = $number;
            for($i=2; $i<=$number; $i++){
                    $this->result*=$i;
            }
            echo "__construct() executed. ";
    }
```

```
    function factorial($number){
            $this->number = $number;
            for($i=2; $i<=$number; $i++){
                    $this->result*=$i;
            }
            echo "factorial() executed. ";
    }

    public function showResult(){
            echo "Factorial of {$this->number} is {$this->result}. ";
    }
}
?>
```

现在，我们使用这个类来创建一个新对象，如代码清单 2-5 所示：

代码清单2-5　使用阶乘类声明对象

```
<?php
include_once("class.factorial2.php");
$fact = new factorial(5);
$fact->showResult();
?>
```

该脚本执行后会得到如下结果：

```
__construct() 执行了. 5 的阶乘为 120.
```

类似构造方法，有一个析构方法，它在做销毁一个对象或相关的工作。

你可以显式地创建命名为 __destruct（）析构方法的方法。这种方法将在你的 PHP 脚本执行结束前执行。为了测试这一点，可以将下面的代码添加到类中：

```
function __destruct(){
    echo " 对象已销毁.";
}
```

当我们再执行该脚本时，会看到析构方法输出的执行结果：

```
__construct() 执行了. 5 的阶乘为 120. 对象已销毁.
```

2.6　实例与多态

实例和多态这两个词汇是面向对象开发的重点，开发者定义类结构，并从中进行数据抽象，最重要的要点是面向对象中的多态概念。

多态表示在一个对象中，它与另一个对象使用的类型相同，如果 B 类是 A 类的后代，A 类的一个方法除了可接收自己实例的参数外，还可以接受子类 B 传递的参数。

1. public

public 修饰的属性或方法可以被其他类在外部访问。

2. protected

protected 修饰的成员变量或方法表示允许对象内部和子类的对象访问。被限定为 protected 的成员变量，只能通过父类本身或子类进行访问或修改。上面的代码就是子类通过继承，可以共享和访问父类中的成员变量以及父类的方法。

现在我们创建另一个文件名 class.extendedemailer.php，使用如代码清单 2-6 所示的代码：

代码清单2-6　创建一个emailer的子类extendedEmailer

```
<?php
class extendedEmailer extends emailer{
    function __construct(){}
    public function setSender($sender){
            $this->sender = $sender;
    }
}
?>
```

下面我们使用该类，即声明该类的对象应用，如代码清单 2-7 所示：

代码清单2-7　使用extendedEmailer扩展类

```
<?php
    include_once("class.emailer.php");
    include_once("class.extendedemailer.php");
    $xemailer = new ExtendedEmailer();
    var_dump($xemailer);
    $xemailer->setSender("webmaster@21cto.com");
    $xemailer->addRecipients("jiang.du@qq.com"); //访问方法
    //给一些方法发送参数
    $xemailer->setSubject("我要找工作");
    $xemailer->setBody("你好，你好?");
    $xemailer->sendEmail();
?>
```

你会发现，我们访问了 extendsendEmail 对象，这实际上继承的是 Emailer 父类的方法。当声明为 protected 的方法时，这意味着非继承方式不能调用。如果我们执行下面的

代码，它会产生一个 PHP 致命错误，如代码清单 2-8 所示：

代码清单2-8　非继承的方法访问protected方法

```
<?php
include_once("class.emailer.php");
include_once("class.extendedemailer.php");
$xemailer = new ExtendedEmailer();
$xemailer->sender = "hasin@21cto.com";
?>
```

我们会得到类似如下的错误信息：

```
<b>Fatal error</b>: Cannot access protected property extendedEmailer::$sender
    in <b>C:\OOP with PHP5\Codes\ch1\test.php</b> on line <b>5</b><br />
```

3. private

private 表示属性或方法被声明为私有，只能由类本身的方法访问，继承该类的子类也是不能访问的。在旧的 PHP 版本的面向对象模式里，类的属于都使用 var 关键字来定义，对于方法，这相当于使用了 public 关键字。

除了以上 3 个关键字，对于成员方法，还有以下 3 个关键字描述，分别为静态（static）、抽象（abstract）和最终（final）方法。

- static 静态方法虽然隶属于某个类，但它不受该类的束缚，不需要声明对象实例就可以直接被外部访问和存取。
- abstract 抽象方法不能直接使用，必须经过实现（使用 implement 关键字）才可以使用。
- final 方法，表示该方法是最终的版本，不能再重新声明，也不能被重写。

在类中也可以使用常量，常量的值是在运行时不能被改变，与变量的区别是，它们以大写字符表示，并且不能使用 $ 美元符。

2.7　类的扩展

在面向对象开发中，它最大的特点之一就是可以扩展一个类，创建一个新的子类。新的子类可以保留所有的父类方法或重写的方法。我们来扩展 emailer 类，重写 sendEmail 方法，以便它可以发送 HTML 邮件，如代码清单 2-9 所示：

代码清单2-9 扩展emailer的HtmlEmailer类

```
<?php
class HtmlEmailer extends emailer{
    public function sendHTMLEmail(){
            foreach ($this->recipients as $recipient){
                    $headers = 'MIME-Version: 1.0' . "\r\n";
                    $headers .= 'Content-type: text/html; charset=iso-8859-1' ."\r
                        \n";
                    $headers .= 'From: {$this->sender}' . "\r\n";
                    $result = mail($recipient, $this->subject, $this->body,
                    $headers);
                    if ($result) echo "HTML Mail successfully sent to{$recipient}
                        <br/>";
            }
    }
}
?>
```

由于这个类扩展了 emailer 类，并引入了新的方法：sendHTMLEmail()，我们仍然可以从其父类的方法调用，如代码清单 2-10 所示：

代码清单2-10 使用发送邮件类

```
<?php
include("class.htmlemailer.php");
$hm = new HtmlEmailer();
//etc....
$hm->sendEmail();
$hm->sendHTMLEmail();
?>
```

如果想访问父类中的方法，我们可以使用 parent 关键字。例如，如果要访问父类中名叫 sayHello() 的方法，可以写成 parent::sayHello()。

上面的代码清单中，没有写任何关键字调用 sendEmai，这表示该方法是从父类 emailer 类的继承。

另外，如果子类中没有重写构造方法，执行时父类的构造方法同样会被调用。

2.8 防止重写

如果你将方法声明为 final 方法，它不能被任何子类覆盖 / 重写。如果不希望有人来重写你的类或方法，都可以声明为 final。我们来看代码清单 2-11：

代码清单2-11　声明为final方法

```
<?php
// base class
class Base {
    final public function testMethod() {
        echo 'This is a final method';
    }
}

class BaseChild extends Base {
    // override testMethod()
    public function testMethod() {
        echo 'Another text';
    }
}
?>
```

如果我们执行上述代码，PHP 会告诉我们一个致命的错误，因为子类试图重写父类的 final 方法。类似于如下方法：

```
Fatal error: Cannot override final method Base::testMethod() in ...
```

2.9　防止被扩展

类似于 final 类，也可以将某个方法声明为 final，即不允许重写该方法。如代码清单 2-12 所示，它被声明为最终的方法，不允许被重写。

代码清单2-12　最终的方法

```
<?php
class SuperClass {
       public final function someMethod(){
          //..方法中的相关代码
       }
}
class SubClass extends SuperClass {
       public function someMethod(){
          //..方法中的相关代码，但是不会被执行
      }
}
?>
```

如果我们执行以上代码，会得到如下类似的错误信息：

```
<b>Fatal error</b>: Class bclass may not inherit from final class
(aclass) in <b>C:\OOP with PHP5\Codes\ch2\class.aclass.php</b> on
line <b>8</b><br />
```

2.10 多态性

多态性是建立几个具体的父类对象的过程。拿上面的例子来说，我们创建了 3 个类，Emailer、ExtendedEmailer 和 HtmlEmailer，如代码清单 2-13 所示：

代码清单2-13 一个类的多次实现

```
<?php
include("class.emailer.php");
include("class.extendedemailer.php");
include("class.htmlemailer.php");
$emailer = new Emailer("info@21cto.com");
$extendedemailer = new ExtendedEmailer();
$htmlemailer = new HtmlEmailer("hasin@somewherein.net");
//判断此对象是否属某个类
if ($extendedemailer instanceof emailer ){
   echo "Extended Emailer is Derived from Emailer.<br/>";
}
if ($htmlemailer instanceof emailer ){
   echo "HTML Emailer is also Derived from Emailer.<br/>";
}
if ($emailer instanceof htmlEmailer ){
   echo "Emailer is Derived from HTMLEmailer.<br/>";
}
if ($htmlemailer instanceof extendedEmailer ){
   echo "HTML Emailer is Derived from Emailer.<br/>";
}
?>
```

注意：我们使用了 instantof 关键字，用它来判断当前对象的父类关系。现在我们执行上面的脚本，会出现类似下面的输出结果：

```
Extended Emailer is Derived from Emailer.
HTML Emailer is also Derived from Emailer.
```

2.11 接口

从长相上看，接口就是一个空类，只包含方法的声明。

所以任何类要实现接口，必须“完成”它里面定义的所有方法。就像一个国家的宪法，所有州县法律是基于此基本法来定义的。你也可以把接口理解成是一个严格的声明。接口有助于在扩展类时严格执行在接口中定义的所有方法。可以通过使用关键字 implements 来实现接口。

为什么需要接口这样的东西呢？

接口相对于一个普通类来说，使用它意味着一个严格的规范。

举个例子，我们在创建一个 Web 应用时，这个应用可能会和不同的数据库有连接和处理操作，可以有 MySQL、Postgral SQL、MariaDB、SQLLite 等。我们现在的开发团队要开发不同的数据库驱动类。

那么会是怎样的策略呢，分配给 3 个人，让他们按自己的风格开发？这当然没问题。但是使用这些类时必须仔细看这些类的方法和类定义，比如有人连接数据库用 conn 命名，有人喜欢用 connect，千奇百怪，美不胜收。对于调用的人，这简直太枯燥、太折磨人，后面也很不好维护。

因此，我们需要严格定义该接口。本例给它有一个固定的名字，driver，然后有两个方法，名为 connect() 和 execute()。实现这个接口需要严格按此规约来“实现”，调用的开发都不必再担心类或方法有变化，写接口类的开发者也可以随时优化内部方法，无须伤及应用。

下面我们创建数据库驱动 driver 接口类，如代码清单 2-14 所示：

代码清单2-14　数据库驱动接口类driver定义

```
<?php
//interface.dbdriver.php
interface DBDriver
{
    public function connect();  //数据库连接方法
    public function execute($sql); //执行SQL方法
}
?>
```

注意到了吗？所有的方法都是空的，这是一个纯接口类。

现在，让我们创建一个 MySQLDriver 类，来试图实现这个接口，如代码清单 2-15 所示：

代码清单2-15　创建MySQLDriver类

```
<?php
```

```
//class.mysqldriver.php
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
}
?>
```

现在，如果我们执行上面的代码，它会提供以下错误，因为 MySQLDriver 类中没有 connect() 和 execute() 方法的具体实现。运行此段脚本就会出现如下错误信息：

```
<b>Fatal error</b>: Class MySQLDriver contains 2 abstract methods
and must therefore be declared abstract or implement the remaining
methods (DBDriver::connect, DBDriver::execute) in <b>C:\OOP with
```

既然 PHP 已经告诉我们了，那我们就挨个方法地实现 DBDriver 抽象类，不能有一个遗漏，如代码清单 2-16 所示：

代码清单2-16　实现DBDriver接口的方法

```
<?php
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
   public function connect()
   {
          //connect to database
   }
   public function execute()
   {
          //execute the query and output result
   }
}
?>
```

我们尝试运行一下这个初步实现的方法。你会得到类似如下的错误信息：

```
<b>Fatal error</b>: Declaration of MySQLDriver::execute() must be
compatible with that of DBDriver::execute() in <b>C:\
ch2\class.mysqldriver.php</b> on line <b>3</b><br />
```

乍一看，我们都已经实现以上相关方法，但是错误消息说 execute() 方法不兼容。我们再看定义的接口，会发现 execute() 方法应该有一个参数。所以我们实现接口类，每一个方法的结构必须与接口定义完全相同。让我们重写 MySQLDriver 类，如代码清单 2-17 所示：

代码清单2-17　完整实现的MySQLDriver类

```
<?php
include("interface.dbdriver.php");
class MySQLDriver implements DBDriver
{
    public $host="localhost";
    public $username="username";    // specify the sever details for mysql
    public $database="database name";
    public $myconn;

    public function connect()
    {
            //connect to database
            $conn= mysqli_connect($this->host,$this->username,$this->password);

                    if(!$conn)// testing the connection
                    {
                            die ("Cannot connect to the database");
                    }

                    else
                    {

                            $this->myconn = $conn;

                            echo "Connection established";

                    }

                    return $this->myconn;
    }

    public function execute($query)
    {
            //execute the query and output result
            $result = mysqli_query($conn,$query);

            return $result;

    }
}
?>
```

以上代码为相对完整的清单，我们还可以再完善接口处的方法逻辑。

2.12 抽象类

抽象类和接口几乎相同，只不过方法体里可以包含函数体内容。抽象类必须是“extended——扩展”，而不是“implements——实现”。因此，如果扩展类一些方法都需要完成，那么你可以定义一个抽象类。我们来看代码清单 2-18：

代码清单2-18 生成HTML报表的抽象类

```
<?php
//abstract.reportgenerator.php
abstract class ReportGenerator{       //声明为抽象类
    public function generateReport($resultArray){
        //生成HTML报表
    }
}
?>
```

在上面的抽象类中，有一个名为 generateReport 的方法，它接收一个多维数组作为参数，然后生成一个 HTML 报表。

那为什么我们声明一个抽象类？因为后面会用到一个通用 DBDriver，它不会影响代码，因为它正在作为一个参数，而不是任何相关的数据库本身只有一个数组。现在我们用这个 MySQLDriver 的抽象类，如代码清单 2-19 所示：

代码清单2-19 使用抽象类并实现MySQLDriver的方法

```
<?php
include("interface.dbdriver.php");  //引入dbdriver接口
include("abstract.reportgenerator.php"); //引用HTML报表抽象类
class MySQLDriver extends ReportGenerator implements DBDriver {
    public function connect() {
        //连接数据库
    }
     public function execute($query) {
        //执行SQL查询并返回
    }
}
?>
```

大家可能注意到，MySQLDriver 扩展了 ReportGenerator 抽象类并实现了 DBDriver 接口。

我们可以使用抽象类，并全部实现上面的例子所示的接口。

类似声明抽象类，你还可以声明抽象方法。当一个方法声明为抽象时，意味着子类

必须重写该方法。

一个抽象的方法不包含任何内容。抽象方法的声明如下所示：

```
abstract public function connectDB();
```

2.13　静态方法和属性

要访问类中的任何方法或属性，我们必须创建一个对象实例（用 new 关键字，比如 $obj = new emailer()），否则就不能访问它们。

但是如果把方法和属性定义为静态，开发者就可以直接访问，不需要创建这个类的任何实例，就像是一个静态成员为该类的全局成员一样。此外，静态属性会保持被分配的最后状态，这在某些情况下非常有用。

你可能会问，为什么要有静态方法？其实静态方法常被用来描述最实用的方法，用来执行一个非常具体的任务，或者返回一个特定的对象（静态属性和方法常用于设计模式）。

还举前面的例子，我们的应用中需要同时支持 3 种数据库，MySQL、PostgreSQL 和 SQLite。现在，我们需要在同一时间使用一个数据库驱动类。

为此，我们设计一个 DBManager 类，它可以实例化任何请求，并返回到调用的脚本中，如代码清单 2-20 所示：

代码清单2-20　DBManager数据库代理类

```
<?php
//class.dbmanager.php
class DBManager
{
    public static function getMySQLDriver(){
            //初始化MySQL驱动对象并返回
    }

    public static function getPostgreSQLDriver(){
            //初始化PostgreSQL驱动对象并返回
    }

    public static function getSQLiteDriver(){
            //初始化SQL Lite驱动对象并返回
    }
}
?>
```

我们如何使用这个类，刚才已经说了，直接访问，访问静态属性和方法，使用：：运算符即可。具体使用方法如代码清单 2-21 所示：

代码清单2-21　访问静态属性和方法

```
<?php
//test.dbmanager.php
include_once("class.dbmanager.php");
$dbdriver = DBManager::getMySQLDriver();  //注意两个冒号的关键字
?>
```

聪明的你可能也注意到了，我并没有创建任何 DBManager 对象的实例，以前是这样的：

```
$dbmanager  = new DBManager()
```

而这里我们直接访问它的方法，使用：：静态引用关键字就可以，有的书里说成是符，具体叫什么看个人喜好。

那么，使用静态方法究竟有多大的好处？很明显，我们只需要一个驱动程序对象，没必要再创建一个新的 DBManager 对象，不用占内存，也能让脚本正常运行。

静态方法通常执行特定任务。

再提醒一下大家，如果是静态方法或属性，就不能使用 $this 来访问了。因为没有实例化类，$this 会出现比较奇怪的事情。那么在类中如何存取静态属性和方法呢，来看代码清单 2-22：

代码清单2-22　类中访问静态属性

```
<?php
//class.statictester.php
class StaticTester
{
    private static $id=0;
    function __construct(){
             self::$id +=1;
    }

    public static function checkIdFromStaticMehod(){
            echo "Current Id From Static Method is ".self::$id."\n";
    }

    public function checkIdFromNonStaticMethod(){
            echo "Current Id From Non Static Method is ".self::$id."\n";
    }
```

```
}

$st1 = new StaticTester();
StaticTester::checkIdFromStaticMehod();

$st2 = new StaticTester();
$st1->checkIdFromNonStaticMethod(); //returns the val of $id as 2
$st1->checkIdFromStaticMehod();
$st2->checkIdFromNonStaticMethod();
$st3 = new StaticTester();
StaticTester::checkIdFromStaticMehod();
?>
```

运行以上脚本后，你会看到输出如下：

```
Current Id From Static Method is 1
Current Id From Non Static Method is 2
Current Id From Static Method is 2
Current Id From Non Static Method is 2
Current Id From Static Method is 3
```

每当我们创建一个新实例，它影响到所有的变量声明为静态实例。使用这种特殊的驱动程序，设计模式“单例”就是使用了这一特点。

2.14　魔术方法

1. __get() 与 __set()

我们可以通过 __get()、__set()、__call() 方法来存取类中没有定义的成员方法和属性。当我们试图写入一个不存在或不可见的属性时，PHP 就会执行类中的 __set() 方法。__set() 方法必须接收两个参数，用来存放试图写入的属性名称和属性值。来看下面的脚本例子，如代码清单 2-23 所示：

代码清单2-23　使用__set与__get魔术方法

```
<?php
    class MyShop {
      private $p = array();

       function __set($name, $value) { //取得属性名称和值
            echo "set::$name:$value <br />";
            $this->p[$name] = $value;
      }
```

```
    function __get($name) { //取得属性名称
        print "get::$name <br />";
        return array_key_exists($name,$this->p) ? $this->p[$name] : null;
    }
}

$shop = new MyShop();
$shop->apple = 2;
$shop->pear = 3;
$shop->pear++;
echo "苹果=". $shop->apple. "<br>";
echo "梨=". $shop->pear. "<br />";
?>
```

执行结果如下：

```
set::apple:2
set::pear:3
get::pear
set::pear:4
get::apple
苹果=2
get::pear
梨=4
```

从以上例子中，我们可以总结出 __get() 和 __set() 方法的功能：在引用该类中不存在的成员变量时，可以调用这两个方法，我们还可以用它们实现错误消息的提示，可以通过这两个方法动态地创建新变量来扩展一个类。

2. __call()

当我们试图调用类中一个不存在或不可见的方法时，PHP 会执行该类中的 __call() 方法。__call() 也必须接收两个参数，用来存放试图调用的方法名称及其参数（参数会被放在一个与该参数同名的数组中），如代码清单 2-24 所示：

代码清单2-24　使用__call方法

```
<?php
  class  callClass  {
    function  __call($method_name,  $parameters)  {
      echo('使用__call尝试调用一个不存在/不可用的成员方法');
      echo('<i>'.$method_name.'</i>');
      echo('<b>  ,  __call()开始调用</b><br>');
      echo('<b>从传递的参数传入parameters数组，内容如下</b><br><pre>');
      print_r($parameters);
```

```
    }
  }
  $obj  =  new  callClass();
  $obj->someMethod(1,9.9,"测试文本");
?>
```

该脚本执行的结果如下：

```
使用__call尝试调用一个不存在/不可用的成员方法someMethod , __call()开始调用
从传递的参数传入parameters数组，内容如下：
Array
(
[0] => 1
[1] => 9.9
[2] => 测试文本
)
```

看了上面例子，相信你应该能够理解 __call() 的含义了。下面我们运行一个实际应用例子，如代码清单 2-25 所示：

代码清单2-25　__call方法应用实例

```
<?php
class MyShop {
    private $obj;

    function __construct($obj) {
        $this->obj = $obj;
    }

    function __call($method, $args) {
        print $method."::".implode($args,",")."\n";
        if (isset($this->obj) && method_exists($this->obj, $method)) {
           return call_user_func_array(array($this->obj, $method), $args);
        }
    }
}

class Calculate {
    private $items = 0;
    function add($num){
        $this->items += $num;
    }
    function sum(){
        return $this->items;
    }
```

```
}

$obj = new Calculate();
$shop = new MyShop($obj);
$shop->add(2);
print $shop->sum();
?>
```

那么，上面的脚本代码执行后的结果是这样的：

```
add::2 sum:: 2
```

3. __sleep() 与 __wakeup()

__sleep() 方法在序列化（serialize）一个实例的时候被调用，__wakeup() 则是在反序列化（unserialize）的时候被调用。

需要注意一点，__sleep() 必须返回一个数组或者对象（一般返回的是当前对象 $this），返回的值将会被用来做序列化的值，如果不返回这个值，则表示序列化失败。这也意味着反序列化不会触发 __wakeup() 事件。

下面我们再来看一个引用序列化的实用例子，如代码清单 2-26 所示：

代码清单2-26　序列化引用实例

```
<?php
class myMagic
{
    public $a='xx';
    public $b='55';
    function __sleep(){
            echo "I am sleepy\n";
            return $this;
    }

    function __wakeup(){
            echo "wake up!\n";
    }
}
$i = new myMagic;
$si = serialize($i);
echo "sleeping now.....\n";
print_r(unserialize($si));
?>
```

4. __toString()

我们先看一个应用实例，如代码清单 2-27 所示：

代码清单2-27　错误应用实例

```
<?php
class Person {
    private $name;
    function __construct($name){
        $this->name = $name;
    }
}
$obj = new Person("Raymond du");
echo $obj;
?>
```

该程序将输出以下的信息：

```
Catchable fatal error: Object of class Person could not be converted to string
    in toString.php on line 9
```

上面这句话提示我们：捕捉到致命的错误，Person 类不能被转换为 String。

PHP 提供一个叫 __toString() 的魔术方法，可以把类的实例转化为字符串，所以对于上面的实例，可以做以下修改，如代码清单 2-28 所示：

代码清单2-28　__toString方法应用实例

```
<?php
class Person {
   private $name;
   function __construct($name){
          $this->name = $name;
   }

   function __toString(){
          return $this->name;
   }
}
$obj = new Person("Raymond du");
echo $obj;
?>
```

以上的代码将输出为：

```
Raymond du
```

__toString() 成员方法调用并打印当前类中构建器中的值。在这个结构中，调用了公共的字符串操作，这样字符串的连接也就形成了一个字符串。

5. __autoload()

在编写面向对象程序时，常规做法是将每一个类保存为一个 PHP 源文件，这样做的好处是很容易找到一个类在什么地方，并且在需要调用某个类的时候，直接使用 include 或者 require 引用到当前文件就可以了。缺点是，我们每次都要包含一些源文件，如果调用的类或函数很多，需要在源代码头部包含一堆 include 或 require 的代码。

__autoload() 方法可以方便地解决这个问题，在一些情况下省却使用 include 或 require 语句的麻烦。

如果我们在类中定义了 __autoload() 方法（一个类中仅能使用一次），而你访问一个类还未定义，则 __autoload() 方法开始将这个类名作为文件名参数来调用该类，如果能够成功引入该类，脚本将继续顺利执行下去，如果没有调用到该类，PHP 引擎则抛出一个 fatal 错误，停止该脚本的执行。

下面的例子显示了如何调用 __autoload() 方法。这个文件名为 MyClass.php，文件内容如代码清单 2-29 所示：

代码清单2-29　MyClass.php脚本内容

```
<?php
class MyClass {
    function printWorld(){
            echo "物有本末，事有始终，知所先后，则近道矣\n";
    }
}
?>
```

general.inc.php 文件的内容如代码清单 2-30 所示：

代码清单2-30　general.inc.php脚本内容

```
<?php
function __autoload($class_name){
    $file = (dirname(__FILE__)  . "/libs/classes/$class_name.php");
    if(!file_exists($file)){
            return false;
    } else {
            require_once($file);
    }
}
?>
```

main.php 文件用于包含上面的脚本，如代码清单 2-30 所示：

代码清单2-31　main.php脚本内容

```
<?php
require_once("general.inc.php");
$obj = new MyClass();
if(is_object($obj)){
    $obj->printWorld();
}else{
    echo "类文件调入失败。";
}
?>
```

上面的脚本例子将显示下列词句：

```
物有本末，事有始终，知所先后，则近道矣
```

词语本意可以忽略，从结果我们可以看到在 MyClass.php 中并未明确指出包含哪个文件，但是由于使用了 __autoload() 方法，把经常使用的类自动引入进来，如 general.inc 文件的内容，所以能显示上述的内容。

需要注意的是，虽然 PHP 对类名的大小写不敏感，但 __autoload() 方法会按你发送给它的类名严格进行大小写匹配。

如果你喜欢大小写混合的方式来命名一个类，那么在源代码内也要保持大小写一致，这样才可能引入你需要的类库。如果不太喜欢这样做，可以用 strtolower() 函数，在类引入之前强制命名为小写的形式。

另外，这里的内容可以再结合本书的 include 内容部分，更好地融会贯通。

2.15　命名空间

在 PHP5.3 之前，我们经常将类命名为 Product_info_price_list 之类，以避免类还有函数的命名冲突和污染。

另外还一个目标，比如我们想在一个项目中使用两个框架，需要我们从这些框架中抽取一些有用的函数或类，因为这些框架有着这样那样的优势，经过挑选后的整合是个大坑，那就是很多类的命名是一样的，重名冲突会相当严重。

从 PHP 5.3 之后，它顺应潮流，推出了命名空间（name space）这样一个概念。如果你写过 Java 或 .NET，那么就能明白，这个概念就是用来解决上面这些命名混乱等问题

的。命名空间可以让我们不用起那么长的名字，可以继续使用较短的名字来命名，也可以省却一些学英语的小烦恼。

PHP 命名空间均声明在文件顶部，适用于所有在该文件中声明的类、方法和常数。

我会重点介绍命名空间对类的影响，这些原则也适合于其他项目。我们举例来说，下面这段代码都包含在名为 Shipping 的命名空间里，如代码清单 2-32 所示：

代码清单2-32　shipping的命名空间代码

```
namespace shipping;
class courier {
   public $name;
   public $home_country;
   public static function getCourier($courier_list){
           return $courier_list;
   }
}
```

正常情况下，如果想实例化一个 Courier 对象，PHP 会在全局命名空间中寻找这个类，由于它已经被定义在 shipping 命名空间里了，结果肯定是找不到的。

正确引用的方法是使用它的全名：shipping\Courier（注意是反斜杠）。

当在全局命名空间里将所有的类整齐放入小命名空间时，这个工作当然是非常棒的，但当要在另一个命名空间的代码中包含类时我们该怎么办？遇到这种情况，只需要使用一个引导命名空间的标识：反斜杠放在类名的前面，表示 PHP 将从命名空间中的顶部开始查找。

因此，在任意命名空间中使用命名空间中的类，可以这样做：

```
namespace Fred;
$courier = new \shipping\Courier();
```

要引用 Courier 这个类，我们需要知道自己在哪一个命名空间中，比如：

- 在 shipping 命名空间中，称为 Courier。
- 在全局命名空间中，称为 shipping\Courier。
- 在其他命名空间中，需要从顶部开始，这样来指代它：\shipping\Courier。

为了突出命名空间的价值，我们可以在一个名为 Fred 的命名空间下声明另一个 Courier 类，而且在代码中两个对象可以使用相同的类名而不报错，只需在顶层命名空间中声明这两个类即可。

当我们想使用两个或更多框架下的类库而不出毛病时，如 Laravel 或 Zend，这些框

架中都可能有一个类叫 Log。可以在命名空间内部再创建命名空间，注意命名空间之间用分隔符就可以。

举个例子，一个网站可能同时具有博客和电子商务的功能，它有类似这样一个命名空间的类结构：

```
Shop
    Products
          products
productCategory
   shipping
          Courier
admin
   user
          User
```

由于 Courier 类位于第 3 层，用命名空间声明时将 shop\shipping 放进一个文件顶部即可。

加上适当的前缀，我们使用通过命名空间操作符替代多个下划线的方法，解决了长类名的问题。PHP 还是允许我们像操作数据表一样把名字进行简写，以指代命名空间，包括在一个文件中使用多个命名空间。

代码清单 2-33 描述了我们刚才所描述的一系列类，如下：

代码清单2-33　使用命名空间

```
use shop\shipping;      //使用哪个命名空间
use admin\user as u;    //命名空间别名
//我们用哪个命名空间的Courier类
$couriers = shipping\Courier::getCouriersByCountry('China');

//浏览用户账号并显示名字
$user = new u\User();
Echo $user->getDisplayName();
```

我们可以看到，因为声明了 shipping 命名空间，可以将嵌套命名空间中的最低一层作为缩写。第二行的命名空间使用，我们将 user 重命名成 u，可以用 u 这个空间名来引用这个类。

以上这些对于我们逐步解决多数特定元素的同名问题大有裨益，你完全可以为这些名字另取一个更有特色的简称来加以区分。

事实上，命名空间被更多地应用于自动加载（Auto Load）功能上，它长得也非常像目录分隔符。相对于 PHP 而言，命名空间是一个新增加的内容，我们需要在类库和框架

两层概念中深入理解它们。

2.16 traits

从 PHP5.4 开始，PHP 实现了代码复用的方法，这个方法被称为 traits，这个功能用来解决 PHP 只支持单继承的问题。

举例来说，我们设计一个网站，它有不同的类，如用户（user）、页面（page）、文章（article）等。当我们开发时，如果有一个能够不去关心对象类型，每个类里都有一个调试方法，用来打印出一个指定对象的信息，那对我们的调试是非常有用的。

比如这个方法是这样定义的：

```
function myVarDump(){
    //打印对象的信息
}
```

方法里的调试逻辑我省略了，你可以自行添加想实现的调试逻辑。

接下来我们需要在每个类中添加粘贴这个方法，但是这样做会造成不必要的代码冗余，而且一旦该方法的定义有更改，后面就需要修改很多东西。

通常情况下，当我们需要在多个不同类中使用同一个方法的时候，可使用 include 包含再静态引用，另外继承是一个不错的解决方案。

在 PHP 中，每个类只能单一继承，即每个类只能从一个父类继承，这样的话就不能为多个类指定同一个通用的父类。解决办法是有的，那就是现在介绍的 traits。这个功能允许我们在不使用继承的情况下为一个类增加方法。

我们创建 traits，要使用 trait 关键字，后面的命名和类名规范是相同的，加上命名和内容定义即可，如下代码清单 2-34 所示：

代码清单2-34　traits的定义

```
Trait examTrait{
    //属性
    function someFunction(){
            //方法内容
    }
}
```

traits 和抽象类、接口一样，不能从 trait 创建一个对象（继承）。我们需要另一个关键字 use 来在一个类中增加一个 trait 用例。代码如下：

```
class someClass{
    use SomeTrait;
    //类的定义内容

}
```

就像在一个 PHP 脚本中使用 include 包含一个外部的 PHP 脚本就能使其马上生效一样，在这里增加一个 use TraitName 语句就可以使这个 trait 的代码在当前类生效。

现在，当我们创建了一个 SomeClass 类型的对象时，这个对象就有了 someFunction() 方法。

```
$obj = new SomeClass();
$obj -> someFunction();
```

在下列的示例程序中，我们将使用 trait 实现上述的调试程序，并且在一个类中使用 trait。要实现这个功能，我们将使用 3 个函数，虽然在前面没有提及这 3 个函数，从函数名中也能看得出它们的各自作用。

下面我们新建一个 PHP 脚本，保存为 tDebug.php。我们将在一个单独脚本中定义 trait，在另一个脚本里引用对象。tDebug.php 的代码清单如下：

代码清单2-35　tDebug.php脚本内容

```
<?php
trait tDebug{
    public function dumpObject(){
    //取得类名称
    $class = get_class($this);

    //取得属性
    $attributes = get_object_vars($this);

    //取得方法
    $attributes = get_object_vars($this);

    //打印头内容
    echo “<h2>关于对象信息$class object</h2>”;

    //打印属性
    echo “<h3>属性</h3>”;
    foreach($attribtes as $k=>$v){
            echo “<li>$k : $v </li>”;
    }
    echo “</li></ul>”;
```

```
    //打印方法
    echo “<h3>方法</h3>”;
    foreach($methods as $v){
            echo “<li>$v </li>”;
    }
    echo “</li></ul>”;
    }//结束方法
}//结束一个trait定义
```

2.17 本章小结

在本章中，我们详细了解了 PHP 的面向对象特性，以及面向对象开发的设计与实例，包括如何在 PHP 中创建类、实现类的封装方法、实现类的定义、调用类的方法、静态方法、类的扩展、重载多态。

另外，面向对象与面向过程各有所长，并非面向过程就会造成代码冗余，绝大多数还是取决于团队。在语言层面，PHP7 中又推出了如闭包、回调等新的特性，值得各位进一步探索。

第 3 章 Chapter 3

PHP 输出缓冲区

在这一部分中，主要向大家介绍 PHP 输出缓冲区的核心技术和最佳实践。

输出缓冲区一直是 PHP 开发者的一个盲点。很多开发者包括我自己以前只是知道这个概念以及它大概怎么用，但对于它的原理却了解不多。

当你阅读完本章，相信可以解决大部分的困惑并了解它的原理，从而有效地解决性能问题。

3.1 系统缓冲区

为了理解更顺畅，我们先了解操作系统的缓冲区。

缓冲区（Buffer），实际是一个内存地址空间。它是用来在存储速度不同步的设备或者优先级不同的设备之间传输数据的区域。通过缓冲可以使进程之间的交互时间等待变小，从而使从速度慢的设备读入数据时，速度快的设备的操作进程不发生间断。比如在一个 4GB 内存的 Linux 系统下，其缓冲区默认大小为 4096 字节，缓冲区在内存中的位置和表现如图 3-1 所示。

图 3-1 表示的是在 Linux 下的内存映射图，我们用 free –m 命令就可以看到不同类型内存区的占用情况。

再如我们在打开文本编辑器（如 VIM 编辑器）编辑文件时，每输入一个字符，操作系统不会立即把这个字符直接写入磁盘，而是先写入缓冲区，当写满时才把缓冲区中的

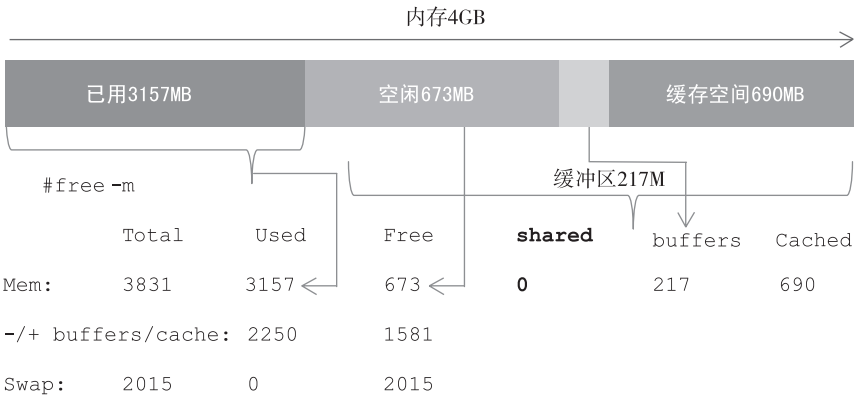

图 3-1 Linux 内存映射图

数据写到磁盘。也就是当内核函数 flush() 被调用时，才会强制把缓冲区中的“脏数据”保存到磁盘中。

通过以上内容，我想你已经了解系统级别的输出缓冲区，接着来看 PHP 输出缓冲区的原理。

3.2 什么是 PHP 输出缓冲区

PHP 的输出流包含很多内容，通常都是开发者要 PHP 输出的文本，这些文本大多是用 echo 语句或 printf() 函数来输出的。

对于 PHP 中的输出缓冲区，我们需要了解 3 点内容。

（1）任何会输出内容的函数都会用到输出缓冲区。

当然这指的是正常的 PHP 脚本，如果开发的是 PHP 扩展，使用的函数（C 函数）可能会直接将输出写到 SAPI 缓冲区层，不需要经过输出缓冲层。


提示 我们可以在 PHP 源文件 main/php_output.h 中了解到这些 C 函数的 API 文档，这个文件提供了很多其他的信息，例如默认的缓冲区大小。

（2）输出缓冲区层不是唯一用于缓冲输出的层，它实际上只是很多层中的一个。输出缓冲区层的行为与你使用的 SAPI（Web 或 CLI）有关，不同的 SAPI 可能有不同的行为。

我们先通过以下图片来看看这些层的关系，如图 3-2 所示。

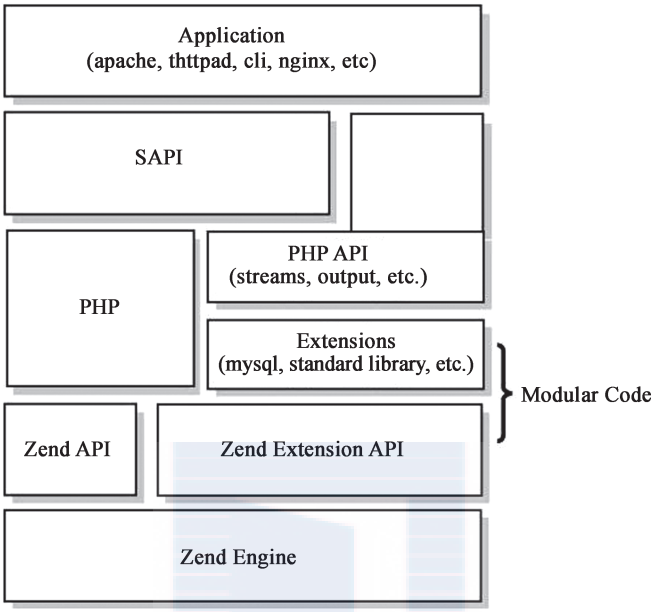

图 3-2　PHP 缓冲逻辑关系图

这张图展示了 PHP 中的 3 种缓冲区层的逻辑关系。

最上端的两层就是我们通常所认识的“输出缓冲区”。

（3）SAPI 中的输出缓冲区。这些都是 PHP 中的层，当输出的字节离开 PHP 进入计算机体系结构中的更底层时，缓冲区又会不断出现（终端缓冲区（terminal buffer）、fast-cgi 缓冲区、Web 服务器缓冲区、操作系统缓冲区、TCP/IP 栈缓冲区等）。

PHP CLI 的 SAPI 有点儿特殊，CLI 也称命令行界面。它会将 php.ini 配置中的 output_buffer 选项强制设置为 0，这表示禁用默认 PHP 输出缓冲区。所以在 CLI 中，默认情况下你要输出的内容会直接传递到 SAPI 层，除非你手动调用 ob_() 类函数。并且在 CLI 中，implicit_flush 的值也会被设置为 1。我们经常会混淆 implicit_flush 的作用，PHP 的源代码已说明一切：当 implicit_flush 被设置为打开（值为 1）时，一旦有任何输出写到 SAPI 缓冲区层，它都会立即刷新（flush，意思是把这些数据写到更低层，并且缓冲区会被清空）。

换句话说，任何时候当你写入任何数据到 CLI SAPI 中时，CLI SAPI 都会立即将这些数据扔到它的下一层去，一般会是标准输出管道，write() 和 fflush() 这两个函数就是负责做这件事情的。关于 CLI 的更多细节请大家参考第 9 章的内容。

3.2.1 默认 PHP 输出缓冲区

如果你使用不同于 CLI 的 SAPI，比如 PHP-FPM，会用到下面 3 个与缓冲区相关的 .php.ini 文件的配置选项：

- output_buffering。
- implicit_flush。
- output_handler。

在搞清楚这几个选项的含义之前，有一点需要先说明一下，不能在运行时使用 ini_set() 函数修改这几个选项的值。这些选项的值会在 PHP 程序启动的时候，还没有运行任何脚本之前解析，所以也许在运行时可以使用 ini_set() 改变它们的值，但改变后的值并不会生效——一切都已经太迟了，因为输出缓冲区层已经启动并激活。我们只能通过编辑 php.ini 文件或者是在执行 PHP 程序的时候使用 -d 选项才能改变它们的值。

默认情况下，PHP 发行版会在 php.ini 中把 output_buffering 设置为 4096 个字节。如果你不使用任何 php.ini 文件（或者也不会在启动 PHP 的时候使用 -d 选项），它的默认值将为 0，这表示禁用输出缓冲区。如果你将它的值设置为“ON”，那么默认的输出缓冲区的大小为 16KB。

你可能已经猜到了，在 Web 应用环境中对输出的内容使用缓冲区对性能有好处。默认的 4k 的设置是一个合适的值，这意味着你可以先写入 4096 个 ASCII 字符，然后再与下面的 SAPI 层通信。并且在 Web 应用环境中，通过 Socket 一个字节一个字节地传输消息的方式对性能并不好。更好的方式是把所有内容一次性传输给服务器，或者至少是一块一块地传输。

层与层之间的数据交换次数越少，性能越好。应该总是保持输出缓冲区处于可用状态，PHP 会负责在请求结束后把它们中的内容传输给终端用户，开发者不用做任何事情。

implicit_flush 已在前面谈论 CLI 的时候提到过。对于其他的 SAPI，implicit_flush 默认被设置为关闭（off），这是正确的设置，因为只要有新数据写入就刷新 SAPI 的做法很可能并非你所希望的。对于 FastCGI 协议，刷新操作是每次写入后都发送一个 FastCGI 数组包，如果发送数据包之前先把 FastCGI 的缓冲区写满会更好。如果你想手动刷新 SAPI 的缓冲区，请使用 PHP 的 flush() 函数。如果你想写一次就刷新一次，可以设置 php.ini 中的 implicit_flush 选项，或者调用一次 ob_implicit_flush() 函数。

因此，我们推荐在 php.ini 中使用的配置如下：

```
output_buffering = 4096
```

```
implicit_flush = false
```

要修改输出缓冲区的大小，应确保使用的值是 4 或 8 的倍数，它们分别对应 32 位和 64 位操作系统。

output_handler 是一个回调函数，它可以在缓冲区刷新之前修改缓冲区中的内容。PHP 的扩展提供了很多回调函数（用户也可以自己编写回调函数）。

- ob_gzhandler：使用 ext/zlib 压缩输出。
- mb_output_handler：使用 ext/mbstring 转换字符编码。
- ob_iconv_handler：使用 ext/iconv 转换字符编码。
- ob_tidyhandler：使用 ext/tidy 整理输出的 HTML 文本。
- ob_[inflate/deflate]_handler：使用 ext/http 压缩输出。
- ob_etaghandler：使用 ext/http 自动生成 HTTP 的 Etag。

缓冲区中的内容会传递给你选择的回调函数（只能用一个）来执行内容转换的工作，所以如果你想获取 PHP 传输给 Web 服务器以及用户的内容，可以使用输出缓冲区回调。有一点也需要提一下，这里说的“输出”指的是消息头（header）和消息体（body）。HTTP 的消息头也是输出缓冲区层的一部分。

3.2.2　消息头和消息体

当你使用一个输出缓冲区（无论是用户的，还是 PHP 的）的时候，你可能想以你希望的方式发送 HTTP 消息头和内容。你知道任何协议都必须在发送消息体之前发送消息头（这也是为什么叫作“头”），但是如果你使用了输出缓冲区层，那么 PHP 会接管这些，而不需要你操心。

实际上，任何与消息头的输出有关的 PHP 函数（header()，setcookie()，session_start()）都使用了内部的 sapi_header_op() 函数，这个函数只会把内容写入消息头缓冲区中。当我们输出内容，如使用 printf() 函数，内容会先被写入到输出缓冲区（可能是多个）。当这个输出缓冲区中的内容需要被发送时，PHP 会先发送消息头，然后发送消息体。PHP 为你搞定了所有的事情。如果你想自己动手，那就只能禁用输出缓冲区，除此之外别无他法。

3.2.3　用户输出缓冲区

对于用户输出缓冲区（user output buffer），我们先通过一个示例来看看它是怎么工

作的，以及你可以用它来做什么。再强调一下，如果你想使用默认 PHP 输出缓冲区层，就不能使用 CLI，因为它已禁用了这个层。下面的这个示例用的就是默认 PHP 输出缓冲区，使用了 PHP 的内部 Web 服务器 SAPI：

```
/* launched via php -doutput_buffering=32 -dimplicit_flush=1 -S127.0.0.1:8080
     -t/var/www */
echo str_repeat('a', 31);
sleep(3);
echo 'b';
sleep(3);
echo 'c';
```

在这个示例中，启动 PHP 时是将默认输出缓冲区的大小设置为 32 字节，程序运行后会先向其中写入 31 个字节，然后进入休眠状态。此时屏幕是空白的，什么都不会输出，跟程序设置一样。2 s 之后休眠结束，再写入一个字节，这个字节填满了缓冲区，它会立即刷新自身，把里面的数据传递给 SAPI 层的缓冲区，因为我们将 implicit_flush 设置为 1，所以 SAPI 层的缓冲区也会立即刷新到下一层。字符串 'aaaaaaaaaa{31 个 a}b' 会出现在屏幕上，然后脚本再次进入睡眠状态。2 s 之后，再输出一个字节，此时缓冲区中有 31 个空字节，但是 PHP 脚本已执行完毕，所以包含这一个字节的缓冲区也会立即刷新，从而会在屏幕上输出字符串 'c'。

从这个示例我们可以看到默认 PHP 输出缓冲区是如何工作的。我们没有调用任何与缓冲区相关的函数，但这并不意味着它不存在，它就存在于当前程序的运行环境中（在非 CLI 模式中才有效）。

接下来开始讨论用户输出缓冲区，它通过调用 ob_start() 创建，我们可以创建多个这种缓冲区（直到内存耗尽为止），这些缓冲区组成一个堆栈结构，每个新建缓冲区都会堆叠到之前的缓冲区上，每当它被填满或者溢出，都会执行刷新操作，然后把其中的数据传递给下一个缓冲区。

```
ob_start(function($ctc){ static$a = 0; return$a++ . '- ' .$ctc . "\n";}, 10);
ob_start(function($ctc){ returnucfirst($ctc); }, 3);
echo"fo";
sleep(2);
echo'o';
sleep(2);
echo"barbazz";
sleep(2);
echo"hello";
/* 0- FooBarbazz\n 1- Hello\n */
```

我们假设第一个 ob_start 创建的用户缓冲区为缓冲区 1，第二个 ob_start 创建的为缓冲区 2。按照栈的后进先出原则，任何输出都会先存放到缓冲区 2 中。

缓冲区 2 的大小为 3 个字节，所以第一个 echo 语句输出的字符串 'fo'（2 个字节）会先存放在缓冲区 2 中，还差一个字符，当第二个 echo 语句输出 'o' 后，缓冲区 2 满了，所以它会刷新（flush）。在刷新之前会先调用 ob_start() 的回调函数，这个函数会将缓冲区内的字符串的首字母转换为大写，所以输出为 'Foo'。然后它会被保存在缓冲区 1 中，缓冲区 1 的大小为 10。

第三个 echo 语句会输出 'barbazz'，它还是会先放到缓冲区 1 中，这个字符串有 7 个字节，缓冲区 1 已经溢出了，所以它会立即刷新，调用回调函数得到的结果为 'Barbazz'，然后被传递到缓冲区 2 中。这个时候缓冲区 2 中保存了 'FooBarbazz'，10 个字符，缓冲区 1 会刷新，同样的先会调用 ob_start() 的回调函数，缓冲区 1 的回调函数会在字符串前面添加行号，以及在尾部添加一个回车符，所以输出的第一行是 'o- FooBarbazz'。

最后一个 echo 语句输出了字符串 'hello'，它大于 3 个字符，所以会触发缓冲区 2 刷新，因为此时脚本已执行完毕，所以也会立即刷新缓冲区 1，最终得到的第二行输出为 '1- Hello'。

因此，使用 echo 函数如此简单的事情，如果牵涉到缓冲区和性能，也是复杂的。因为在你的应用中，要注意使用 echo 输出内容的大小，如果缓冲区配置与输出内容相似，那么性能会比较优良，如果缓冲区配置小于输出内容，需要在应用中对输出的内容做切分处理。

3.3　输出缓冲区的机制

有必要先了解一下 PHP 缓冲区的前世今生。从 PHP 5.4 版开始，整个缓冲区层就都被重写了（该模块由 Michael Wallner 完成）。之前的缓冲区代码比较糟糕，很多功能都没有，而且有很多 BUG。重写后的缓冲区层不仅架构设计得更好，代码也更加整洁，添加了一些新特性，与 PHP 5.3 版的不兼容问题也变少了。

其中最值得称赞的一个特性是我们自己开发 PECL 扩展时，可以声明属于自己的输出缓冲区回调方法，这样可以与其他 PECL 扩展做区分，避免产生冲突。在此之前，这是不可以的，如果要开发使用输出缓冲区的扩展，必须先搞清楚所有其他提供了缓冲区回调的扩展可能带来的影响。

下面的脚本示例展示了如何通过注册回调函数来将缓冲区中的字符转换为大写。示例代码写得可能不是很好，但可以满足我们的实验目的。

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif
#include "php.h"
#include "php_ini.h"
#include "main/php_output.h"
#include "php_myext.h"
static int myext_output_handler(void **nothing, php_output_context *output_
context)
{
    char *dup = NULL;
    dup = estrndup(output_context->in.data, output_context->in.used);
    php_strtoupper(dup, output_context->in.used);
    output_context->out.data = dup;
    output_context->out.used = output_context->in.used;
    output_context->out.free = 1;
    return SUCCESS;
}
PHP_RINIT_FUNCTION(myext)
{
    php_output_handler *handler;
    handler = php_output_handler_create_internal("myext handler", sizeof("myext
        handler") -1, myext_output_handler, /* PHP_OUTPUT_HANDLER_DEFAULT_SIZE
        */128, PHP_OUTPUT_HANDLER_STDFLAGS);
    php_output_handler_start(handler);
    return SUCCESS;
}
zend_module_entry myext_module_entry = {
    STANDARD_MODULE_HEADER,
"myext",
    NULL, /* Function entries */
    NULL,
    NULL, /* Module shutdown */
    PHP_RINIT(myext), /* Request init */
    NULL, /* Request shutdown */
    NULL, /* Module information */
"0.1", /* Replace with version number for your extension */
    STANDARD_MODULE_PROPERTIES
};
#ifdefCOMPILE_DL_MYEXT
ZEND_GET_MODULE(myext)
#endif
```

此部分的 C 代码请感兴趣的读者自行解析。

3.4　输出缓冲区的陷阱

有些 PHP 的内部函数也使用了输出缓冲区，它们会叠加到其他的缓冲区上，这些函数会填满自己的缓冲区然后刷新，或者是返回里面的内容。比如 print_r()、highlight_file() 和 highlight_file::handle() 都是此类。你不应该在输出缓冲区的回调函数中使用这些函数，这样会导致未定义的错误，或者至少得不到你期望的结果。

同样的道理，当 PHP 执行 echo、print 时，也不会立即通过 TCP 输出到浏览器，而是将数据先写入 PHP 的默认缓冲区中。我们可以理解为，PHP 自己有一套输出缓冲机制，在传送给系统缓存之前建立一个新的队列，数据须经过该队列。当一个 PHP 缓冲区写满以及脚本执行逻辑需要输出时，脚本会把里面的数据传输给 SAPI 浏览器，如图 3-3 所示。

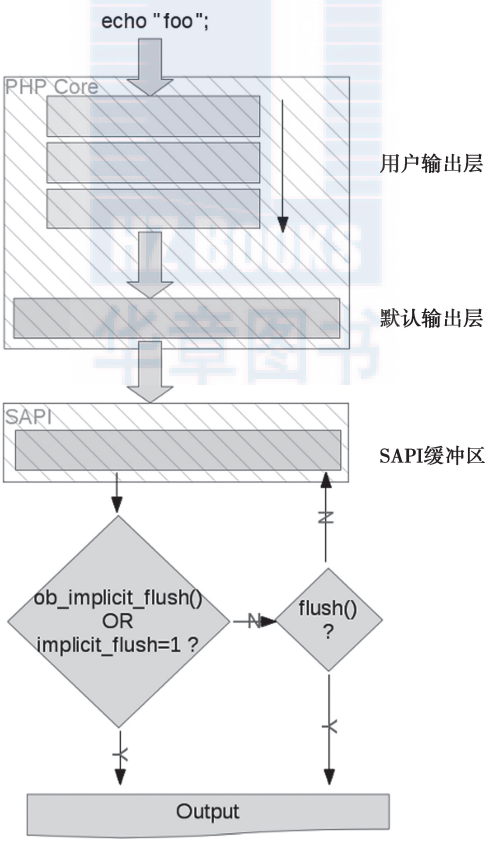

图 3-3　数据缓冲示意图

数据会依次写到这几个地方，分别是：echo/print → PHP 输出缓冲区→ SAPI 缓冲区→ TCP 缓冲区→浏览器。

3.5 输出缓冲区实践

PHP 缓冲区是默认开启的，它的默认参数在 php.ini 配置文件中，值是 4096 字节。在其中找到 output_buffering 配置参数来修改 PHP 缓冲区的大小。

开发者也可以在脚本中通过 ob_start() 函数手动处理 PHP 缓冲区机制。这样即便输出内容超过了配置参数的大小，也不会把数据传输给浏览器，ob_start() 将 PHP 缓冲区空间设置到足够大，只有脚本执行结束后或调用 ob_end_flush() 函数，才会把数据发送给浏览器。

我们编辑 php.ini 配置文件，对 output_buffering 值进行修改并做如下测试。当 output_buffering 修改为 4096 时，输出较少数据，让它小于一个 PHP 缓冲区。代码如下：

```
for ($i = 0; $i< 10; $i++) {
    echo $i . '<br/>';
    sleep($i + 1); //
}
```

执行后你会发现，它不会像常规逻辑每隔几秒就有输出，而是直到脚本循环结束后，才会一次性输出。这种情况在脚本处理结束之前，浏览器界面会一直保持空白，这是由于数据量太小，输出缓冲区没有写满。写数据的顺序：echo 语句输出到 PHP 缓冲区、TCP 缓冲区、浏览器。

接下来我们再修改 output_buffering=0，仍输出较少数据，但实际数据已经大于 PHP 缓冲区。代码如下：

```
for ($i = 0; $i< 10; $i++) {
    echo $i . '<br/>';
    flush();  //通知操作系统底层，尽快把数据发给客户端浏览器
    sleep($i + 1); //
}
```

该脚本的结果与刚才一定不一致，因为将缓冲区的容量设置为 0，即禁用 PHP 缓冲区机制。

这时我们会在浏览器看到断断续续的间断性输出，而不必等到脚本执行完毕才看到输出。这是因为，数据没有在输出缓存中停留。写数据的顺序依次是 echo 输出到 TCP

缓冲区，再输出给浏览器。

我们再把参数修改为 output_buffering=4096，输出数据大于一个缓冲区。此例中不调用 ob_start() 函数。

准备一个 4KB 大小的文件或者使用 dd 命令在 shell 下创建一个文件：

```
$dd if=/dev/zero of=f4096 bs=4096 count=1
```

使用如下代码进行验证：

```
for ($i = 0; $i< 10; $i++) {
    echo file_get_contents('./f4096') . $i . '<br/>';
    sleep($i +1);
}
```

可以看到，程序响应还没结束（HTTP 连接并未关闭），就可以看到间断性输出，浏览器界面不会一直保持空白。尽管启用了 PHP 输出缓冲区机制，但依然会间断性输出，而不是一次性输出，这是因为 PHP 缓冲区空间不够用，每写满一个缓冲区，数据就会发送到客户端浏览器。

和上例参数一样，即 output_buffering=4096，输出数据大于一个 PHP 缓冲区。这次我们调用 ob_start()，代码如下：

```
ob_start();        //开启PHP缓冲区
for ($i = 0; $i< 10; $i++) {
    echo file_get_contents('./f4096') . $i . '<br/>';
    sleep($i + 1);
}
ob_end_flush();
```

等到服务端脚本全部处理完，响应结束才会看到完整的输出。输出间隔时间很短，以至于感受不到停顿。在输出之前，浏览器一直会保持空白，等待服务器端数据。这是因为，PHP 一旦调用了 ob_start() 函数，就会将 PHP 缓冲区扩展到足够大，直到 ob_end_flush 函数调用或者脚本运行结束才发送 PHP 缓冲区中的数据到客户端浏览器。

可以通过 tcpdump 命令监控 TCP 的报文，来观察一下使用 ob_start() 和没有使用它的区别。

以下是未使用 ob_start() 函数的输出：

```
12:30:21.499528 IP 192.168.0.8.webcache > 192.168.0.28.cymtec-port: .ack 485
    win 6432
12:30:21.500127 IP 192.168.0.8.webcache > 192.168.0.28.cymtec-port: .
    1:2921(2920) ack 485 win 6432
```

```
12:30:21.501000 IP 192.168.0.8.webcache > 192.168.0.28.cymtec-port: .
    2921:7301(4380) ack 485 win 6432
```

使用 ob_start() 函数的输出是类似以下的内容：

```
12:36:06.542244 IP 192.168.0.8.webcache > 192.168.0.28.noagent: .ack 485 win 6432
12:36:51.559128 IP 192.168.0.8.webcache > 192.168.0.28.noagent: . 1:2921(2920)
    ack 485 win 6432
12:36:51.559996 IP 192.168.0.8.webcache > 192.168.0.28.noagent: . 2921:7301(4380)
    ack 485 win 6432
12:36:51.560866 IP 192.168.0.8.webcache > 192.168.0.28.noagent: . 7301:11681(4380)
    ack 485 win 6432
12:36:51.561612 IP 192.168.0.8.webcache > 192.168.0.28.noagent: . 11681:16061(4380)
    ack 485 win 6432
```

与上面的输出对比可以看到，数据报文的时间间隔明显不同。没有使用 ob_start() 时，时间间隔比较大，等待 4s 左右就把缓冲区中的数据发送出去了。数据没有在 PHP 缓冲区中停留过长时间，就将数据发送给了浏览器。这是因为 PHP 缓冲区很快被写满了，不得不把数据发送出去。

启用 ob_start() 后则不同，发送数据包给客户端，几乎是同一时间发出去的。可以推断，数据一直在 PHP 缓冲区中保存，直到调用了 ob_end_flush() 才把缓冲区中的数据发送给客户端浏览器。

我们一起总结缓冲区机制，以加深理解。

ob_start 激活 output_buffering 机制。一旦激活，脚本不再直接输出给浏览器，而是先暂时写入 PHP 缓冲区。

PHP 默认开启 output_buffering 机制，通过调用 ob_start() 函数把 output_buffering 值扩展到足够大。也可以通过 $chunk_size 来指定 output_buffering 的值。$chunk_size 默认值是 0，表示直到脚本运行结束后，PHP 缓冲区中的数据才会发送到浏览器。若设置了 $chunk_size 的大小，则表示只要缓冲区中数据长度达到了该值，就会将缓冲区中的数据发送到浏览器。

可以通过指定 $ouput_callback 参数来处理 PHP 缓冲区中的数据，比如函数 ob_gzhandler()，将缓冲区中的数据压缩后再传送给浏览器。

ob_get_contents() 函数是获取一份 PHP 缓冲区中的数据拷贝，这是一个重要的函数。请看以下示例：

```
<?php
ob_start();
```

```
?>
<html>
<body>
today is <?php echo date('Y-m-d h:i:s'); ?>
</body>
</html>
<?php
$output = ob_get_contents();
ob_end_flush();

echo '<!output>'.$output;
?>
```

以上脚本运行后，查看源代码，会出现两段相同的 HTML，后者就是通过 ob_get_contents() 函数取得缓冲区里的内容。

ob_end_flush() 与 ob_end_clean() 这两个函数都会关闭输出缓冲。

不同的是，ob_end_flush() 只是把 PHP 缓冲区中的数据发送到客户端浏览器，而 ob_clean_clean() 将 PHP 缓冲区中的数据删除，但不发送给客户端。ob_end_flush() 调用之后，PHP 缓冲区中的数据依然存在，ob_get_contents() 依然可以获取 PHP 缓冲区中的数据拷贝。

3.6　输出缓冲与静态页面

大家都知道静态页面的加载速度快，不用请求数据库。用户看到 .html 都会觉得速度快，如代码清单 3-1 所示就是一个生成静态页面的脚本代码：

代码清单3-1　生态静态页面

```
echo str_pad('', 1024);//使缓冲区溢出
ob_start();//打开缓冲区
$content = ob_get_contents();//获取输出缓冲区的内容
$f = fopen('./index.html', 'w');
fwrite($f, $content);//将从缓冲区取得的内容写入文件
fclose($f);
ob_end_clean();//清空并关闭缓冲区
```

我们还会在一些模板引擎和页面文件缓存中看到 ob_start() 函数被使用。在一些知名的开源项目（如 Wordpress、Drupal、Smarty 等）中，都可以发现它的踪影。下面从 Drupal 应用中抽取的一个代码片断，如代码清单 3-2 所示：

代码清单3-2 使用ob_start()函数

```
function theme_render_template($template_file, $variables) {
    if (!is_file($template_file) { return ""; }
    extract($variables, EXTR_SKIP);
    ob_start();
    $contents = ob_get_contents();
    ob_end_clean();
    return $contents;
}
```

值得一提的是，函数 flush() 是把数据直接输出到系统缓冲区，需要和 ob_flush() 函数相区分。

如果你熟悉 Wordpress，经常会在主题目录的 header.php 看到类似的代码：

```
<!DOCTYPE html>
<htmllang="en">
<head>
<meta charset="UTF-8" />
<title>Buffer flushing in action</title>
<linkrel="stylesheet" type="text/css" href="styles.css" />
<linkrel="shortcut icon" href="favicon.ico" />
</head>
<?php
// flush the buffer
flush();
?>
<body>
```

从代码里看到，只有一个 flush()，它所在的位置就是告诉浏览器哪一部分的缓存需要更新，即页面头部以上部分需缓存。

再请看 Wordpress 的部分代码，如下：

```
functionmyplugin_alter_settings_general() {
   // check to see if we're loading the options-general page
   global $parent_file;
   if ( $parent_file != 'options-general.php' ) return;

   // turn on the output buffer and attach the callback
   ob_start('myplugin_general_callback');
}
add_action('admin_head','myplugin_alter_settings_general');

function myplugin_general_callback($data) {  //回调函数
```

```
    // alter $data as I see fit
    return $data;
}
```

可以看到，在缓冲区开启时，加入自己的回调方法 myplugin_general_callback。

3.7　内容压缩输出

内容压缩输出就是把输出到客户端浏览器的内容进行压缩，有点儿像把文件压缩或 ZIP 或 Tar 格式的包。

从理论和实践双重角度来看，对于服务器端和用户端都会有好处。

服务器端，可以降低对服务器出口带宽的占用，提升带宽的利用率，单位带宽可以服务更多的用户请求；降低 Web 服务器（如 Nginx、Apache、Tomcat 等）处理文本时引入的开销，比如内存和 CPU 的使用率，提升服务器的利用率。用户端，可以减少网络传输延时对用户体验的影响；降低浏览器加载页面内容时占用的内存，有利于改善浏览器的稳定性。

我们在 PHP 脚本里使用输出缓冲区加入压缩输出功能，代码如下：

```
<?php
ob_start('ob_gzhandler');  //使用gz格式压缩输出
print "My content\n";
ob_end_flush(); ?>
?>
```

使用 ob_start（'ob_gzhandler'），将内容以压缩方式输出，表示只压缩当前脚本与缓冲区，对其他脚本没有影响。

PHP 还提供另外一种压缩方式，即在 php.ini 中修改：

```
zlib.output_compression = On
```

这样输出时所有页面都以 zlib 的压缩方式输出。在此仅列出这种方法，我不建议使用这种方法，如果只针对某个页面压缩输出，请使用第一种方法。

如果两者混合使用是毫无意义的，只会额外地消耗 CPU 性能，让它压缩已经压缩好的内容。

从代码层面讲，可以使用 PHP 将 HTML、CSS、JS 中的空格和制表符全部去除，这样对提升性能有帮助。

基于实践，使用 PHP 的压缩方法效果并不十分理想。通常的做法是放在 Web 服务器

端，比如 Apache 启用 deflate、Ngnix 使用 gzip 的方式都比 PHP 端压缩效果要好得多。

3.8 本章小结

输出缓冲区就像一张网，它会把所有从 PHP“遗漏”的输出包起来，然后把它们保存到一个大小固定的缓冲区里。当缓冲区被填满时，里面的内容会刷新（写入）到下一层（如果有的话），或者是写入下面的逻辑层：SAPI 缓冲区。开发人员可以控制缓冲区的数量、大小以及在每个缓冲区层可以执行的操作（清除、刷新和删除）。

输出缓冲允许第三方库和应用框架（比如 Laravel、Symfony 等先进的 PHP 框架）开发者完全控制它们自己输出的内容，比如把它们放到一个全局缓冲区中处理。对于任何输出流的内容（如数据压缩）和任何 HTTP 消息头，PHP 都会以正确的顺序发送。

使用输出缓冲能够有效节省带宽，比如图片、字体、CSS、JavaSript 等前端内容。特别是现在的前端框架也越来越大，让它使用户的反应速度更快，从而更有效地提高系统性能。