HZ BOOKS
华章科技

Web开发
技术丛书

PHP与MySQL高性能应用开发

杜江 著

PHP and MySQL
High-Performance Applications Development

作者拥有15年研发经验，资深PHP专家和架构师，曾担任赶集网和今日头条技术总监，好乐买和正和岛的CTO

围绕高性能、可扩展性、可伸缩性、可靠性等与PHP应用性能相关的主题展开，同时还涉及PHP编程思想、底层原理、编程技巧、开发规范等重要内容

机械工业出版社
China Machine Press

PHP与MySQL高性能应用开发

PHP and MySQL
High-Performance Applications Development

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：北京市荣盛彩色印刷有限公司 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

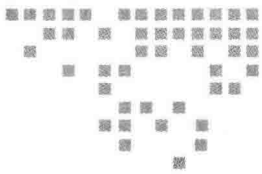
感谢机械工业出版社杨福川、高靖雅和李艺，以及曾经并肩战斗的朋友，是你们的鼓励才能使本书得以展现给各位。PHP 由 PHP 开发小组和众多的 PHPer 共建。同样，本书也得到了很多同仁的支持，在此一并致谢！

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。

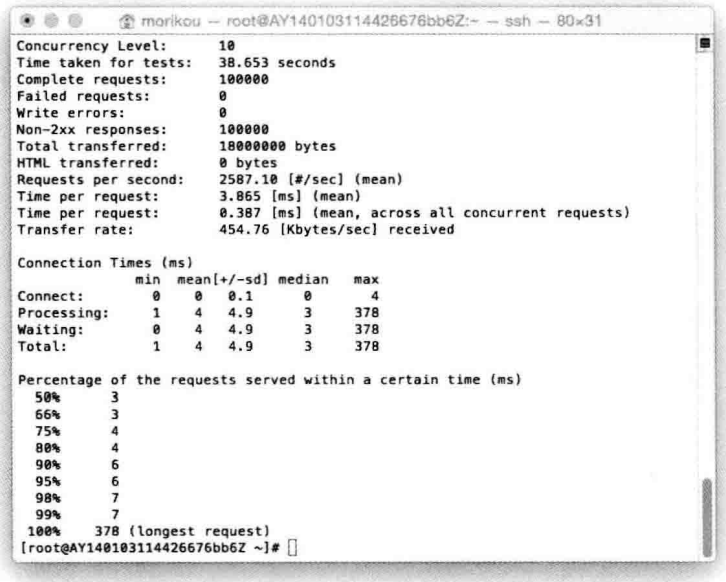


图 1-1　使用 ab 测试并发请求结果

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：

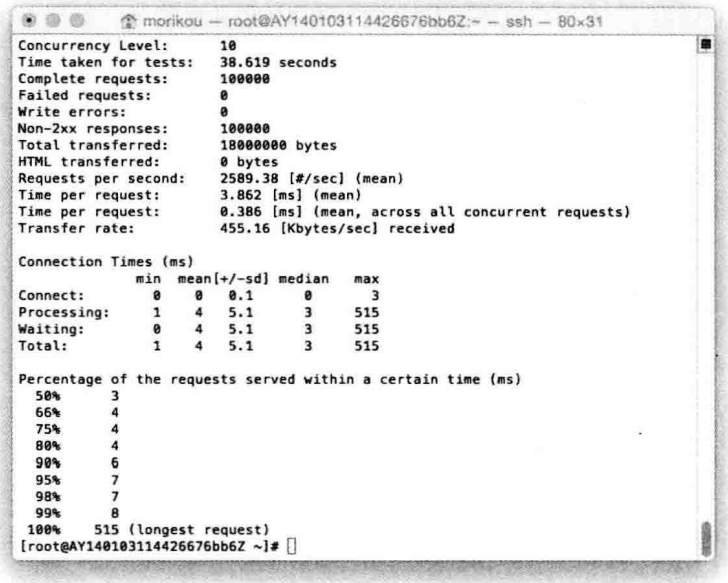


图 1-2 使用 ab 测试 require() 函数的性能

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。

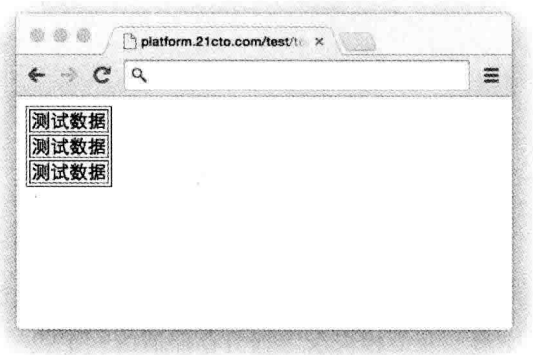


图 1-3 通过函数来打印内容

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。

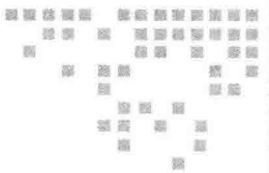
在如今的 Web 应用中，大量使用了缓存技术，它可以降低编码侧的性能问题。但还是需要开发者在写程序时不要偷懒，多留心、留意——其实程序员与工程师的区别也在这里。

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

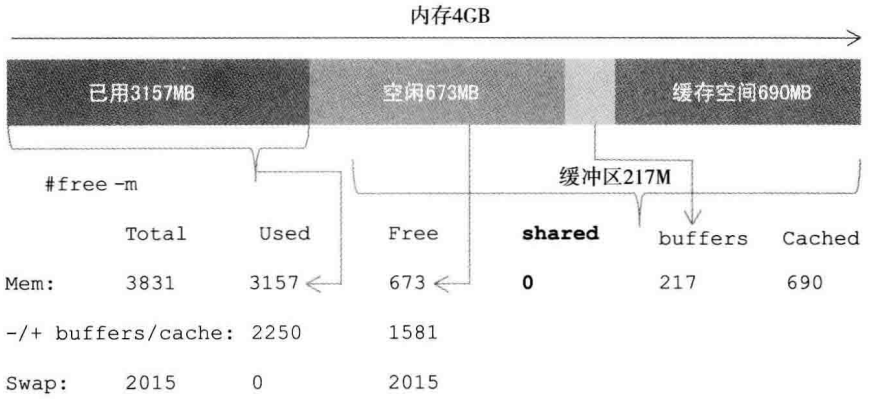


图 3-1 Linux 内存映射图

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。

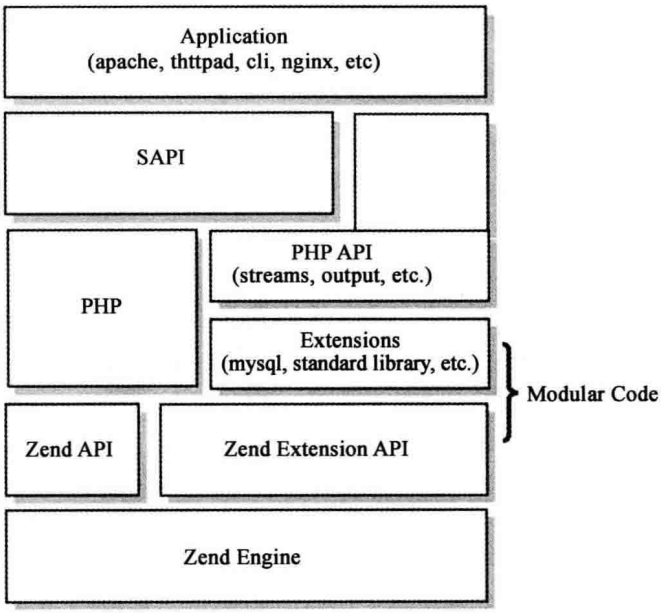


图 3-2　PHP 缓冲逻辑关系图

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 '0- 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。

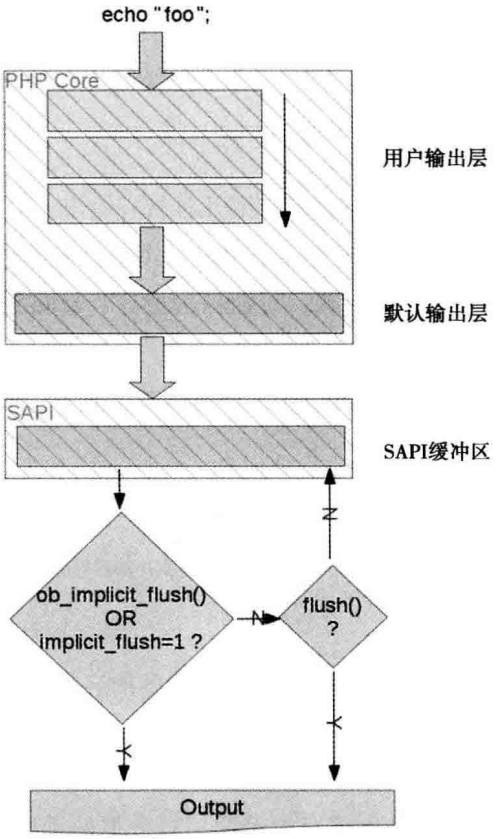


图 3-3　数据缓冲示意图

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。

第 4 章 Chapter 4

PHP 缓存技术

随着网站或应用的流量不断增长，需要我们对代码的性能优化提升，以便能够承载更大规模的数据并发请求。开发者需要关注的核心是提升性能，如管理和使用 OpCode 缓存，利用 memcache、Redis 把缓存放在不同的分布式服务器或云主机上。

可以对网站系统架构及服务器数量制定预备战略，要考虑性能，也要考虑扩展性。在本章中，我们一起讨论缓存相关技术，包括页面缓存、静态化和 OpCode 缓存技术。

4.1 关于缓存

大家知道，HTTP 是个无状态协议，也就是说对于页面处理时没有“记忆”能力。但是对于数据库驱动的网站，需要保存用户登录状态、交互及页面参数传递等信息。

因此在 Web 开发中，无论是使用何种语言来开发，都会用到 Session（信息保存在服务器端，用键值来访问）、Cookie（通常为保存在客户端浏览器中的 Session 的键值）及数据库（物理保存应用程序数据，永久存储）这些技术来保存这些状态信息。

PHP 创始人 Rasmus Lerdorf 在设计 PHP 时，也同时发明了“完全不共享的架构”（Shared Nothing Archictecture）。也就是服务器端的 Session 可以保存在文件、数据库或内存缓存里，客户端 Cookie 保存当前 SessionID 或浏览器 URL 中，带一个加密字符串用来查询 Session 状态就可以了解当前会话状态，由此就可保证网站中用户的持续会话，用

户在使用网站时有持久的会话以及良好的体验。

除了保存用户会话状态外，还有很重要的一点，当网站处于高并发、大流量的状况下时，如果每次请求页面都查询数据库，这样就给数据库的负载增加了压力。所以对于大型互联网应用，对缓存的使用和部署至关重要。大部分网站的操作是读（Read），写（Write）操作很少，而缓存技术正是避免从数据库中读取数据，可以大大减轻服务器的负载。

缓存实现的基本原理是将数据库查询结果以字符串序列化形式保存到磁盘文件中，打开时再反序列化，这样的效率会高于 MySQL 数据库查询，特别是多表连接查询时会特别明显。

那么关于缓存，什么样的数据适合用缓存，要注意哪些问题，有几点规则总结如下：

（1）只缓存很少变化的数据。看似很平常、很简单的一句话，道理也显而易见。它告诉我们，缓存至少要保存一段时间，无论它是 10 分钟、1 小时、1 天还是 1 星期。

如果数据不经常变化，使用缓存是个较好的解决方案，如果数据经常更新，使用缓存则没有必要。

（2）缓存和安全性。不要因为使用了缓存却带来安全性问题。如果缓存中包含用户信息，就更需要关切此类问题。比如多用户系统，由于不同的用户权限不同，缓存中的数据有可能是高权限用户的数据片断，而当前用户看的不是当前权限应该得的内容，导致越权操作，比如删除等功能，引发应用较大的安全性问题。

（3）良好的缓存系统设计。大部分的缓存系统设计，让开发者只需负责插入数据，无须关心处理和验证。因此，缓存系统应该具备调试功能，比如获取数据是否过期，检测某个数据是否已缓存，清空或删除某个片断，当缓存出现问题时的自我修复，以及适时的开关闭功能，等等。当然优秀的缓存系统性能要快，而且还要有较高的命中率。

（4）备份与持久化。缓存的易用机制让我们有时候忘了缓存里的数据会被覆盖或破坏，这在一些很复杂敏感的环境时尤其重要。缓存只是一个暂存数据，对于重要的数据应该及时到文件系统或数据库等持久化保存。

（5）确定缓存过期时间。缓存保存的是暂时的数据，定期刷新，有的数据可能保存的日期更长久一些，但始终都会设定一个过期日期，然后会自动在数据池中删除旧数据。比如 Memcached 和 APC 都有一个 TTL 值，在超过 TTL 时间后就立即清除数据。

（6）基于文件的缓存。保存数据最方便的方法就是保存在文件系统中。

比如在 PHP 中 Session 的默认方式就是保存在文件系统中，当用户发起一个 Session Cookie 时，服务器会在文件系统上加载 Session 数据。

基于文件系统缓存的优点是：PHP 内置文件系统函数，无须额外的扩展或模块，这种文件系统的效率和功能，取决于开发者如何让文件系统写得更高效。

基于文件系统的缓存使用 var_export() 或序列化函数就可以构建。

它也存在一定的缺点，如果网站是多台服务器的集群，在一台服务器存储的文件不能或不太容易被另一台服务器读取；文件缓存也受机器本身的计算能力以及 I/O 速度限制；基于文件的缓存存取时，必须使用文件锁定机制，防止两个并发的线程同时写入一个文件；缓存自动过期需要系统额外关注。

了解了缓存的规则，下面我们就开始了解如何通过缓存和不同的缓存机制来减少网站负载，优化和提高性能。

4.2　文件缓存与静态页面

大多数的网站内容更改并不多，有的页面可能几周甚至更长时间也不会变化。

正常情况下，当用户访问一个网站时，会请求 PHP 脚本去查询数据库的内容，然后再输出到浏览器。比如我们有一个新闻网站，网络编辑每天都会做新闻更新，这些文章会保存到 MySQL 数据库中，以方便网站编辑更新或查询操作。

这个网站有一个名为 news.php 的脚本的内部功能如下：

- 连接 MySQL 数据库。
- 取得最新的文章。
- 文章按时间排序。
- 读取模板页面。
- 输出 HTML 页面给浏览器。

结合该脚本功能，当网站被访问时将产生一系列处理，过程如图 4-1 所示。

按照一般的网站请求量来讲，如果访客只有十几人，即便同时在线 1 小时，对网站的性能消耗微不足道，但如果有 5000 个以上访客在线，那服务器的负载就成倍数增加，当查询或写操作数据库过多时，会在数据库端产生 IO “瓶颈”，表面的情况就是系统响应变得缓慢。

这种问题就需要用缓存技术来解决。当 PHP 脚本运行一次后，即立即转储为缓存文件，下次当用户请求相同的页面时，若数据库内容没有更新，则立即发送缓存的内容给浏览器。

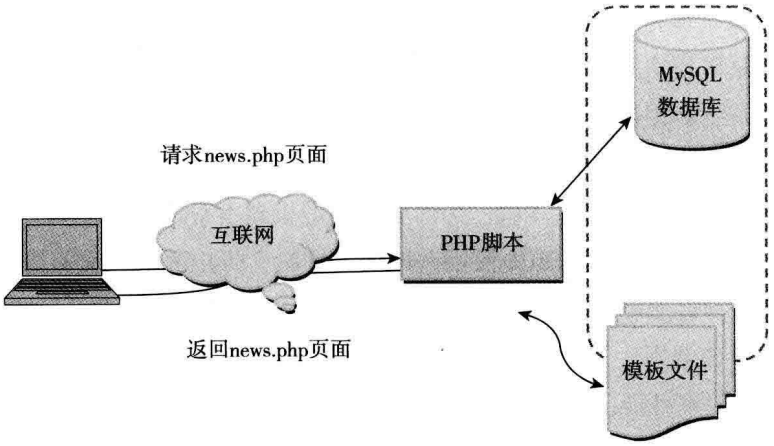


图 4-1　未使用缓存的动态页面请求

图 4-2 描述了在页面请求时缓存的结构。

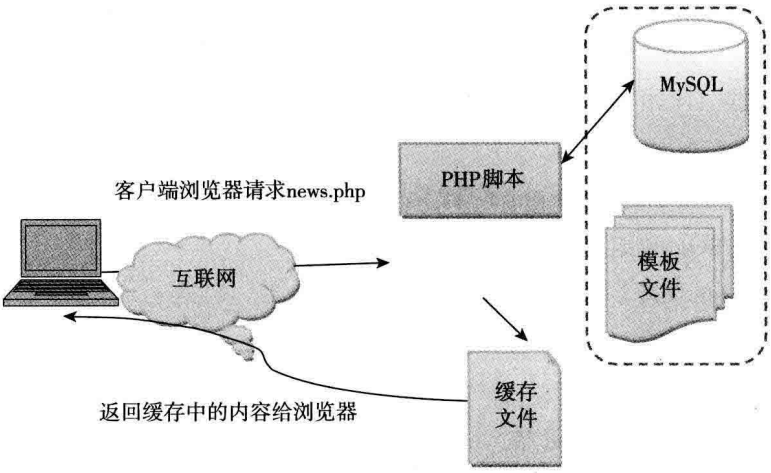


图 4-2　使用缓存的动态页面请求

从图 4-2 中可以看到，当使用缓存后，MySQL 数据库和模板都不再参与处理，Web 服务器发回的是缓存的文件内容，这个请求会在很短的时间内完成，给用户的感觉是网站响应会快很多，在服务器端也大大减少了查询数据库和脚本解析时间，由此便降低了系统开销。常见的文件级别缓存，在模板引擎 Smarty、ADODb、Discuz 等库或产品被广泛应用。

4.3　页面静态化

大多数成功网站都有个共同点，它们都保持相对的简洁风格，没有过多的华丽修饰。产品基本原则是简单、易用，像熟悉的 Google 或百度网站界面，但是简单的界面并不意味着后端没有使用复杂的技术给用户提供更好体验。在本节中，我们将了解从数据库动态取得数据然后生成静态的 HTML 页面。

静态 HTML 页面在一些方面具有一定优势，如加载快、容易编写、服务器负载小，带宽成本降低，而且搜索引擎也喜欢静态页面地址。大多数用户对于静态页面的地址，无论是真的静态页面还是重写的地址，给人的感觉是速度变快，当然这是一个心理范畴。

但是动态网页也具备自己的优势，如自动刷新内容（如新闻、私信、消息等），适合在与用户交互较多的应用，如在线表单查询、数据输出等场景中使用。

我们将两者的优点做一个结合，来个两全其美之策。比如已经有了一个动态网站，内容保存在 MySQL 中，但想在前端访问的是静态 HTML 页面，而且让它在一定时间自动更新。

该功能的实现如代码清单 4-1 所示：

代码清单4-1　build_static.php

```
<?php
// 开始缓冲区
ob_start();

$cachefile = "cache/static.html";
$cachetime = 10*30;  //设置缓存更新时间
if(file_exists($cachefile)&&time()-$cachetime< filetime($cachefile))
{
   include($cachefile);
   echo "<!-- published at ".date("H:i",filetime($cache_file))." by Think Creative
       CMS -->\n";
   exit();
}

echo '<!-- 此处是要生成静态的PHP脚本文件 -->';

$fp = fopen($cachefile,'w');
//将输出缓冲区的内容写到文件
fwrite($fp,ob_get_contents());
// 关闭文件
fclose($fp);
```

```
//将缓冲区内容立即输出到浏览器
ob_end_flush();
?>
```

首先，ob_start() 函数用来启动缓冲区，接下应加入要生成静态页面的 PHP 代码文件内容。然后我们使用 fopen() 函数将内容写到文件中，整个生成过程完成。

该脚本会在 cache 目录下生成一个名为 static.html 的缓存文件。注意，在 Linux 系统下，该文件夹和文件应具有可写（666 或 777）权限。

执行静态文件生成时，需要手工干预或者置于页面以 <script src='build_static.php'></script> 或 <img src='build_static.php' height=1 widht=1> 这种类似标签，在访问所在页面时触发执行。

那么有没有方法能定期自动运行呢？我们可以将该脚本放在 Linux 系统下的 cron 或 Windows 的计划任务中运行。如果想在 24 小时内生成页面，我们可以去掉 cache_time 的代码判断。脚本代码如代码清单 4-2 所示：

代码清单4-2　build_static_cron.php在cron中生成静态文件

```
<?php
// 开始缓冲输出
ob_start();
?>
<!--这里是静态文件的内容-- >
//要生成的PHP文件内容在这里
<?php
$cachefile = "cache/static.html";
//打开static文件并准备写入
$fp = fopen($cachefile, 'w');
//将输出缓存的内容写到文件
fwrite($fp, ob_get_contents());
// 关闭文件
fclose($fp);
//输入内容到浏览器
ob_end_flush();
?>
```

这样，静态页面会随着你设置的 cron 时间来定期生成一个 static.html 的静态文件。静态页面非常适合于内容不是经常更新的情况，因为不用经过 PHP 和 MySQL 等动态处理，因此速度和性能比较高，这也就是新浪、搜狐等新闻门户网站使用静态页面的原因。

4.4　数据级别缓存

前面提到过，静态页面并非屡试不爽的灵药，它也有一些缺点，全页静态化需要占用更多的存储空间，有时候无法预计某一页面的更新频率过高，会使这部分的硬盘 IO 开销更大。

对于一些交互并不是特别高，但是负载又很大的应用，就需要考虑使用缓存这一机制：

- 数据缓存是在全动态的低效与全静态占用大量存储空间之间寻找一个平衡点。
- 数据缓存可在最新的或是比较活跃的数据上使用，在节约存储空间的前提下，尽量提高数据缓存的使用效果。

比如在一个内容型网站，针对列表页（首页、栏目页），及近期的内容页进行缓存，带有分页的列表页，可只缓存前几页的分页。比如论坛，由于访问和更新很快，需要对一部分内容缓存，部分有更新的内容进行手动缓存。

下面我们开始了解缓存的一些机制。

在做文件存储中，所有有意义的数据都需一个键值（Key）来标识，并且它在系统中必须是唯一的，可以使用不同的命名空间并使用 MD5 加密的标识来保存，也可以将类名关键字名称与 ID 合并，作为键值名称。

比如用户管理类叫作 My_Auth, 并且所有用户都使用 id 作为标识，比如保存的缓存数据的内容为 "My_Auth:users:12345"，'12345' 就是用户 id。

我们来看代码清单 4-3 的内容：

代码清单4-3　cache_class.php自定义缓存类

```
<?php
 class Cache_Filesystem
 {
   // 缓存写保存
   function set($key,$data,$ttl)
   {
     //打开文件为读/写模式
    $h = fopen($this->get_filename($key),'a+');
    if (!$h) throw new Exception('Could not write to cache');

    flock($h,LOCK_EX); // 写锁定，在完成之前文件关闭不可再写入

    fseek($h,0); //到该文件头
```

```
    //清空文件内容
    ftruncate($h,0);

    // 根据生存周期$ttl写入到期的时间
    $data = serialize(array(time()+$ttl,$data));
    if (fwrite($h,$data)===false) {
      throw new Exception('Could not write to cache');
    }
    fclose($h);

}

// 读入缓存数据，如果未取出返回失败信息
function get($key)
{
  $filename = $this->get_filename($key);
  if (!file_exists($filename)) return false;
  $h = fopen($filename,'r');
  if (!$h) return false;
  // 文件读写锁定
  flock($h,LOCK_SH);
  $data = file_get_contents($filename);
  fclose($h);

  $data = @unserialize($data);
  if (!$data)
        {
          // 如果反序化失败，则彻底删除该文件
          unlink($filename);
          return false;
  }
  if (time() > $data[0])
       {
          // 如果缓存已经过期，删除该文件
          unlink($filename);
          return false;
     }
     return $data[1];
  }

   function clear( $key )
   {
     $filename = $this->get_filename($key);
     if (file_exists($filename)) {
         return unlink($filename);
```

```
        } else {
            return false;
        }
    }

    private function get_filename($key)
    {
        return ini_get('session.save_path') . '/s_cache' . md5($key);
    }
}
?>
```

在上节的代码中，我们用 file() 函数来打开文件，在本节中的 get 方法中更换了 file_get_contents() 函数来打开缓存文件，这是因为后者打开文件的速度要快于前者。

另外需要对已保存的缓存的 TTL（缓存过期时间）时间进行指定。我们可以在打开文件之前，使用 filetime() 函数检测文件最后修改的时间。

按照缓存的规范，过期的 Cache 要删除，请参见代码中的 clear() 方法。另外，我们将缓存文件保存在 php.ini 设置的 Session 会话目录中，你也可以修改为其他可写的目录中。

接下来，我们将该类与 MySQL 查询结合，将查询数据保存到缓存中，每 10 分钟更新一次，如代码清单 4-4 所示：

代码清单4-4　结合MySQL数据库的缓存类

```
<?php
// 创建新对象
$cache = new Cache_Filesystem();

function getUsers()
{

  global $cache;

  // 自定义一个缓存key唯一标识
  $key = 'getUsers:selectAll';

  //检测数据是否已经缓存
  if (!$data = $cache->get($key)) {
     // 如果没有缓存，则获取新数据
      $db_host = 'localhost';
      $db_user = 'root';
      $db_password = '';
```

```
        $database = 'php_adv';
        $conn = mysql_connect( $db_host,$db_user,$db_password );
        mysql_select_db( $database, $conn );

        //执行SQL查询
        $result = mysql_query("SELECT * FROM users");
        $data = array();

        // 将获取到的数据放入数组$data中
        while($row = mysql_fetch_assoc($result)) {
            $data[] = $row;
        }

        // 保存该数据到缓存中，生存周期为10分钟
        $cache->set($key,$data,600);
    }
    return $data;
}

$users = getUsers();
?>
```

本例中使用原生的 MySQL 函数库连接，是因为大多数人都会知道这些函数。为了方便，我们将连接等信息也写在了函数里，你也可以使用 PDO 或其他数据抽象层。

现在已经掌握了文件级别缓存的实现，这样可以让 PHP 在运行时更快速，另外我们还需要在系统编译 PHP 时加入缓存机制，这被称为 OpCode 缓存。

4.5 OpCode 缓存

OpCode 全称 Operation Code，意为操作码。由于 PHP 是解释型语言，当解释器执行 PHP 脚本时会解析脚本代码，将它们生成可以直接运行的中间代码，称为 Zend OpCode，它类似于 Java 的 ByteCode 或 .NET 的 MSL。

1. PHP 脚本执行顺序与 Zend OpCode

众所周知，Zend 引擎（Zend Engine，由 Zend Technologies 开发）是 PHP 的编译引擎和执行引擎，当它在执行一段 PHP 脚本时，会做以下 4 个步骤的动作。

（1）Scanning (Lexing)：扫描，将 PHP 代码转换为语言片段。

（2）Parsing：解析，将 Tokens 转换成简单而有意义的表达式。

（3）Compilation：编译，将表达式编译成 OpCode。

（4）Execution：顺次执行 OpCode，每次一条，执行完刷新内存后销毁。

其过程如图 4-3 所示。

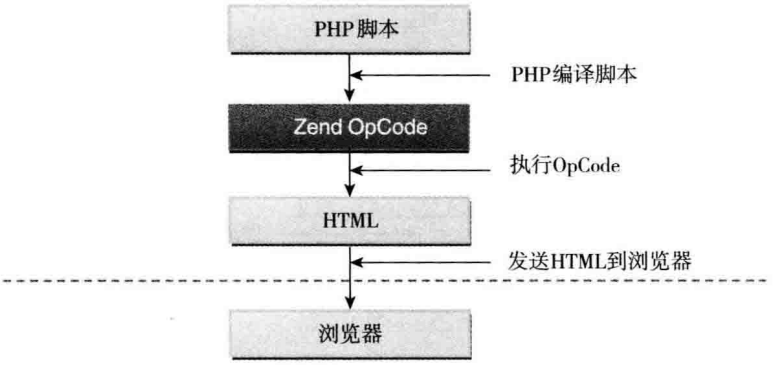


图 4-3　PHP 脚本编译与执行过程

Nginx 或其他 Web 服务器把 HTTP 请求转发给 PHP-FPM，PHP-FPM 再把请求交给某个 PHP 子进程处理，PHP 进程找到 PHP 脚本后执行，把脚本编译为 OpCode 后生成响应。我们从图中看到 PHP 脚本被编译为 Zend Opcode 后生成内容，然后被发送到浏览器客户端。

如果每次请求一个 PHP 脚本都要编译一次 Zend OpCode，然后执行字节码，就会消耗很多资源。如果每次 HTTP 请求 PHP 都必须不断解析、编译和执行 PHP 脚本，消耗的资源更多。如果有一个工具能缓存预告编译好的字节码，减少应用的响应时间，降低系统资源压力，这当然就是我们想要的方式——字节码缓存。

字节码缓存的共通特性就是能够存储预先编译的 Zend OpCode，使用 OpCode 缓存后，当请求一个 PHP 脚本时，不用再读取、解析和编译 PHP 代码。PHP 解释器会从内存中读取预先编译好的字节码，立即执行。这样就能节省很多时间，极大提升应用的性能。

本节就是讨论这些工具，包括独立的扩展，如 Zend Optimizer/Zend OpCache、APC、eAccelerator、ionCube 和 XCache。

2. Zend Optimizer

Zend 提供了核心引擎，于是它又很顺势地推出一款叫作 Zend Optimizer 的免费工具，用来提供实时的 PHP 代码优化。当我们安装过 Zend Optimizer 后，它将在 PHP Zend 引擎的编译器和执行器之间运行，也对就是替换了前面提到的第 2 步和第 3 步，并负责对已编译的脚本进行代码优化，以便交给执行引擎，使执行速度更快，如图 4-4 所示。

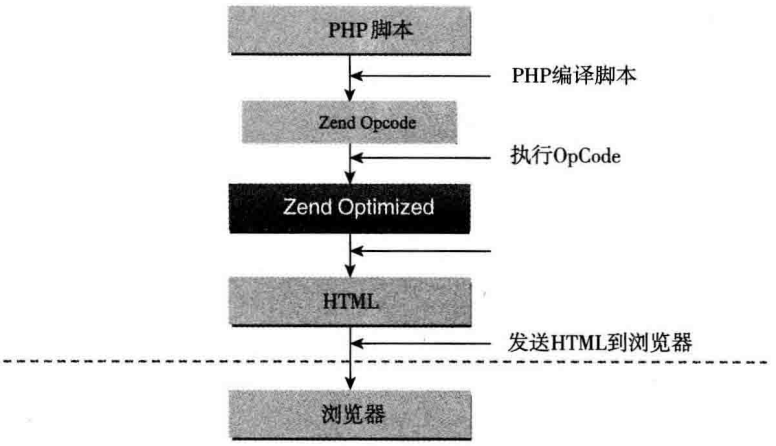


图 4-4 使用了 Zend Optimizer 的 PHP 编译与执行过程

和 Zend Optimizer 类似，Zend 还提供了一款叫作 Zend Guard 的工具，用来加密 PHP 脚本，加密后的脚本必须在已经安装 Zend Optimizer 的服务器环境中才能运行，这也是很多虚拟主机供应商都安装 Zend Optimizer 的原因之一。

目前 Zend Optimizer 的产品推出策略稍有不同，它的 Linux、FreeBSD 及 Windows 的版本推出时间有些不一样。例如 Linux、FreeBSD 和 MacOS 版本均是 3.3.9，而 Windows 版本是 3.3.3，很明显 Windows 32 版本的发布要稍晚一些。

提示 在实际运营环境上，开发者不必追求一些新版本的工具，由于某个软件的版本比较新，而与之匹配的软件则有可能没有更新，则会产生兼容性的问题，导致性能不佳或运行错误，使用稳定的版本是个靠谱的方案。

如果你的 PHP 版本用的是 PHP5.4 以下，那么还可以用 Zend Optimizer。如果到 PHP5.5 后，这个工具已经改名，升级叫作 Zend OpCache，前且内置在 PHP 核心中，无须再安装。

3. Zend Opcache

在 PHP5.5 以后，Zend OpCache 虽然被内置，但默认没有启用，需要显式指定启用 Zend OpCache。

如果是自己编译的 PHP 运行环境，需要在 configure 命令时包含如下：

```
--enable -opcache
```

编译好，须在 php.ini 文件中指定 Zend OpCache 的扩展库的所在路径。例如：

```
zend_extension=/usr/lib64/php/modules/opcache.so
opcache.enable=1
opcache.enable_cli=1
opcache.memory_consumption=128
opcache.interned_strings_buffer=8
opcache.max_accelerated_files=4000
opcache.validate_timestamps =1
opcache.revalidate_freq=60
opcache.fast_shutdown=1
```

重启 PHP FPM，使用 phpinfo 函数查看，确认 Zend OpCache 是否正常工作，如图 4-5 所示。

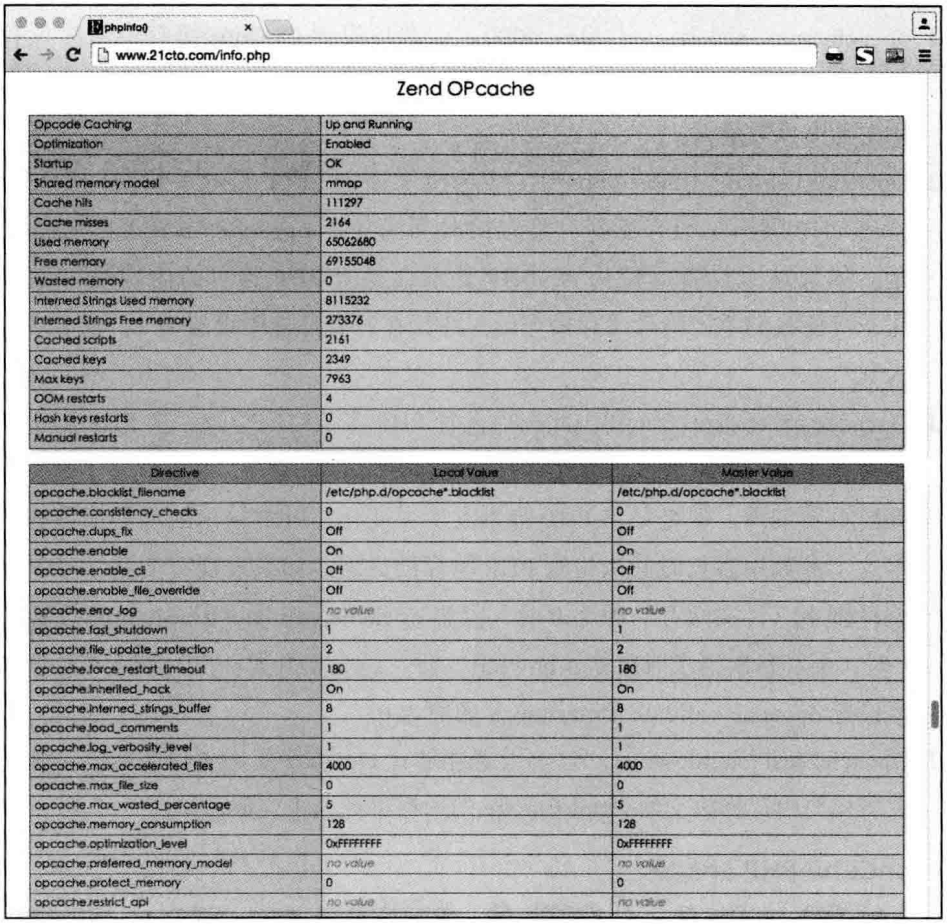


图 4-5　Zend OpCache 的参数设置

下面我们分别解释一下 OpCache 扩展的配置参数。

- opcache.memory_consumption=64。此选项为 OpCode 缓存分配的内存数量，单位为 M。分配的内存量应该能够保存应用中所有 PHP 脚本编译后得到的 OpCode。
- Opcache.interned_strings_buffer=16。此选项用来存储驻留字符串的内存量，单位为 M。PHP 解释器在背景状态下找到相同的字符串实例，保存在内存中，如果再次使用相同的字符串，PHP 解释器会使用指针处理，以保证节省内存。在默认情况下，PHP 驻留的字符串会在各个 PHP 进程中隔离。该选项能够让 PHP-FPM 进程池中的所有进程共享字符串存储，可以让 PHP-FPM 进程之间引用驻留字符串，从而节省更多内存。
- opcache.max_accelerated_files=4000。此选项设置 OpCode 缓存中最多保存多少个 PHP 脚本。该值可以是 200～100 000 的任何数字。注意这个数值一定要比应用中的文件数量大。
- Opcache.validate_timestamps=1。该选项设置为 1 时，经过一段时间后，PHP 会检查 PHP 脚本是否有更新。检查的时间间隔由 opcache.revalidate_rreq 选项指定。如果这个设置的值为 0，PHP 就不会检查 PHP 脚本的内容是否发生了变化，需要我们手工清除缓存操作码。建议在开发环境中设置为 1，生产环境设置为 0。
- Opcache.revalidate_freq=0。此选项设置 PHP 多长时间检查一次 PHP 脚本的内容是否有变化。单位为秒。缓存有好处也存在不好的地方，好处就是不是每次请求都要重新编译，但是如果 PHP 脚本发生了变化，PHP 输出的就是旧内容了。所以这个参数用多长时间来检查 PHP 缓存的更新，如果发现有变化，PHP 会重新编译脚本，再次缓存。值为 0 时，仅当 opcache.validate_timestamps 设置的值为 1 时，会在每次请求时重新验证 PHP 文件，适合于开发一次。在生产环境时，需要设置 opcache.validate_timestamps 的值为 0。
- Opcache.fast_shutdown=1。这个设置能够让 OpCode 使用更快的退出步骤。把对象析构与内存释放交给 Zend Engine 的内存管理器来完成。

4. ionCube PHP Loader

ionCube PHP Loader 是一个 Zend 扩展，需要安装完 Zend Optimizer 后方可再安装使用。

它是英国人 Nick Lindridge 创建的 ionCube 公司（http://ioncube.com/）开发的 ionCube 系列产品之一，该公司有一款叫作 ionCube Encoder 的 PHP 脚本加密工具，与 Zend 公司的 Zend Guard 产品功能相同，用以将源代码化的脚本加密为不可识别的二进制文件。

由于 ionCube PHP Loader 的加密算法与 Zend Guard 不同，不仅支持期限、注册码等加密方式，还支持对 IP、MAC 地址等复杂的加密算法。另外，对于 ionCube 来说，不仅可以加密带有 PHP 标记或源码的 PHP 文件，还可以对非 PHP 文件（以文本方式保存的文件）进行加密操作，如 XML、JS、CSS 等，因此在被防破解方面表现卓越。如果要运行 ionCube Encoder 加密的 PHP 脚本，就需要安装 ionCube PHP Loader，它必须在 Zend Optimizer 之前运行，如图 4-6 所示。

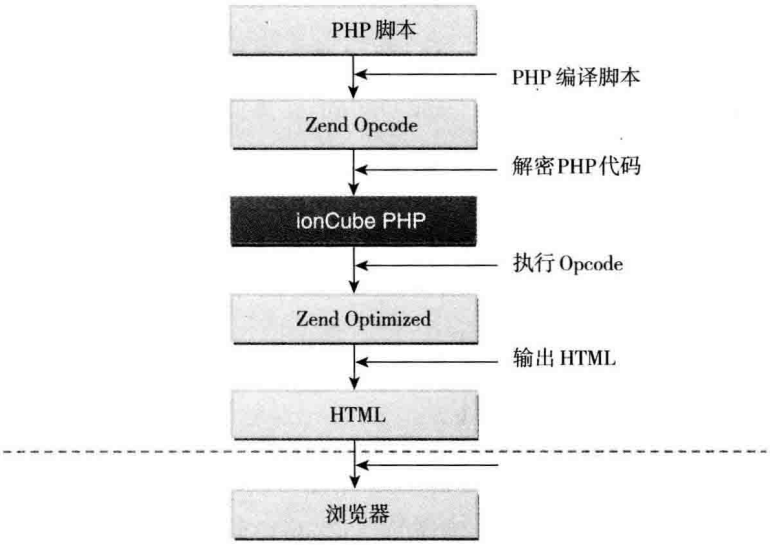


图 4-6　使用了 ionCube Encode 加密的 PHP 编译与执行过程

刚才我们提到过，ionCube PHP Loader 需要在 Zend Optimizer 之前运行，因此在 php.ini 文件中，需要修改如下：

```
;*****************************************************************************
; Zend & Extensions
;*****************************************************************************
[Zend]
; *** ionCube Loader
zend_extension=/usr/local/Zend/lib/ioncube_loader_fre_5.2.so
; *** Zend Optimizer
zend_extension_manager.optimizer=/usr/local/Zend/lib/Optimizer-3.3.3
```

```
zend_extension_manager.optimizer_ts=/usr/local/Zend/lib/Optimizer_TS-3.33
zend_optimizer.version=3.3.3
zend_extension=/usr/local/Zend/lib/ZendExtensionManager.so
zend_extension_ts=/usr/local/Zend/lib/ZendExtensionManager_TS.so
```

安装完成后，即可以看到 ionCube Loader 已经加载运行，如图 4-7 所示。

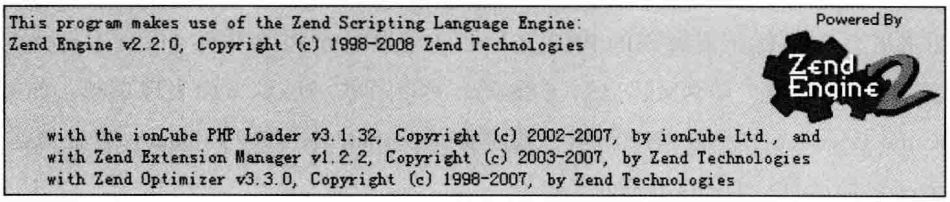


图 4-7　安装完成 ionCube PHP Loader

通过之前内容，我们能看得 Zend Optimizer 是一个实时代码优化与代码解密的工具包，而 ionCube PHP Loader 的功能和 Zend Optimizer 类似。

Zend Optimizer 对性能的提升均有一定作用，而 ionCube 仅对于服务器上有被 ionCube Encoder 加密的脚本运行，如果没有此类脚本则不必安装。

接下来，我们开始讨论对性能提供最大的 Opcode 缓存工具，了解它们的特性以及配置方面的具体内容。

4.6　OpCode 缓存管理工具

目前主流的 OpCode 缓存管理工具主要有 APC、xCache、eAccelerator 及 ionCube PHP Accelerator 等，它们可以将 PHP 编译的 OpCode 中间码保存到缓存中。

这样每次有页面请求的时候，就不需要重复执行 4 步的编译步骤，从而能大幅提高 PHP 的执行速度。这些工具会使用系统内存和硬盘来保存 OpCode 缓存，内存缓存比硬盘缓存的速度快很多，不管是放在内存还是磁盘上，存储的字节码都会快于原始未经处理的 PHP 脚本，因此应用程序的性能会提高很多。

另外缓存工具会根据使用频率，将常用的 OpCode 缓存保存在内存中，其余的保存在磁盘上，从而提高了 PHP 脚本的执行加载时间，降低了服务器的负载。

由于编译一个 PHP 脚本还需要磁盘 IO 访问和大量的 CPU 计算，启用 OpCode 缓存的服务器始终会快于没有使用 OpCode 缓存的服务器，如图 4-8 所示。

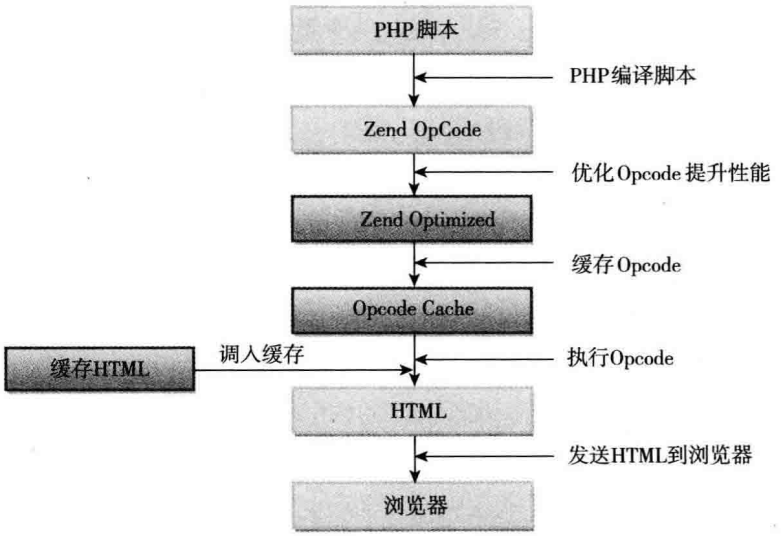


图 4-8　使用了 Opcode 缓存的 PHP 执行过程

4.6.1　使用 APC

1. APC 的安装配置

APC，全称为 Alternative PHP Cache，是较早的 PHP 缓存工具之一。它曾经是 community connect 公司的产品，2005 年由 Yahoo! 开发团队接管并负责技术支持，现在已经是 PECL 库中的一部分。

PHP APC 提供两种缓存功能，编译缓存和用户数据缓存，即缓存 OpCode，称为编译缓存（apc_compiler_cache）。同时它还提供一些接口用于 PHP 开发者将用户数据驻留在内存中，称为用户数据缓存（apc_user_cache）。

在 Linux 下安装 APC 需要使用源代码编译安装。首先下载源代码：

```
#cd /usr/local/src
#wget http://pecl.php.net/package/APC/3.0.19
```

接着使用 pecl 命令开始安装：

```
#pecl install apc
```

刚才的命令操作会把临时文件写到服务器的 /tmp 目录中，需要安装清理该目录：

```
#mv /tmp/APC-3.0.19.tgz /usr/local/src/
```

定位和查找 php.ini 配置文件。我们使用 VI 编辑器打 php.ini 文件：

```
#vi /usr/local/lib/php.ini
```

打开文件后，我们找到以下代码：

```
;Windows Extensions
;Note that MySQL and ODBC support is now built in, so no dll is needed for it.
;
;extension=php_bz2.dll
```

添加 APC 配置信息如下：

```
extension="apc.so"
apc.enabled=1
apc.shm_segments=1
apc.shm_size=128
apc.ttl=300
apc.user_ttl=300
apc.num_files_hint=1024
apc.mmap_file_mask=/tmp/apc.XXXXXX
apc.enable_cli=1
```

最后需要重启 Nginx 或 Apache，可自行建立一个 phpinfo.php 查看 APC 缓存是否已经出现，查看是否安装成功。

2. APC 函数与管理

APC 还提供了一组函数，供开发者调用和查看缓存，常见的函数说明见表 4-1。

表 4-1 APC 缓存函数族

函数名称	说明
apc_cache_info()	返回缓存信息
apc_clear_cache()	清除 APC 缓存内容。默认 (无参数) 时，只清除系统缓存，要清除用户缓存，需用 user 参数
apc_define_constants (string key,array constants [, bool case_sensitive])	将数组 constants 以常量加入缓存
apc_load_constants (string Key)	取出常量缓存
apc_store (string key, mixed var [, int ttl])	在缓存中保存数据。3.0 版本以前的 APC 的函数名称为 apc_add
apc_fetch (string key)	获得 apc_store 保存的缓存内容
apc_delete (string key)	删除 apc_store 保存的内容

下面我们来调用 APC 函数，如代码清单 4-5 所示：

代码清单4-5　向缓存中添加数据

```
// 向缓存中增加数据
function cacheWrite( $sKey, $mData, $iTtl )
{
  if( function_exists( 'apc_store' ) ) { // 如果安装了APC
    return apc_add( $sKey, $mData, $iTtl );
  }
  return false;
}
//读取APC缓存
function cacheRead( $sKey ) {
  if( function_exists( 'apc_fetch' ) ) { //如果安装了APC
    return apc_fetch( $sKey );
  }
  return false;
}

//调用
$sKey = md5($sQuery); //查询返回的数据
if( ($mData = cacheRead( $sKey ) ) === FALSE ) {
  // 信息未被缓存
  $mData = functionThatMakesUpTheData(); //取得数据
  cacheWrite( $sKey, $mData, 600 ); // 开始被缓存，10分钟后更新
}
function cacheClear() {
    if (apc_clear_cache() && apc_clear_cache('user')){ //清空所有缓存
            print '<p>All Clear!</p>';
    }else{
            print '<p>Clearing Failed!</p>';
      }
    print '<pre>'; print_r(apc_cache_info()); print '</pre>';
}
```

前面提到过，APC 的缓存分为系统 OpCode 缓存和用户数据缓存两种类型，下面分别来说明。

（1）系统 OpCode 缓存。系统 OpCode 缓存是自动使用的，指 APC 把 PHP 文件源码的编译结果缓存起来，然后在再次调用事先对比时间标记，如果未过期，则使用缓存代码运行，默认缓存时间为 3600 s（1 小时）。目前对 APC 的性能测试一般指的是这一层缓存。

（2）用户数据缓存。用户数据缓存是由用户在编写 PHP 代码时用 apc_store、apc_

fetch等函数读取和写入的。这一层数据是保存在内存中的。

（3）APC的配置参数。当apc.stat参数设置为0时，将关闭OpCode文件状态（file-stat）跟踪操作；当参数为1时，APC会检查PHP执行之后缓存版本的更改；当脚本已修改数据时，缓存将立即被刷新。

因此关闭文件状态跟踪时，如果一个文件或数据更新，而缓存的内容不一定同步更新。此选项虽然可以提高性能，但因为PHP文件修改后不会立即反映到缓存中，会一直等到被清除或重新启动服务器为止。这是对于文件变化不大时的一个理想状态。

但在开发环境时，PHP文件会经常变更，如果设置为0，我们无法马上知道修改后的效果。另外，file-stat指令对于大多数文件系统是很快的，所以不要指望提升速度有多大。

另外，apc.ttl和apc.user_ttl参数用于指定系统OpCode文件与用户数据文件的缓存生存时间，单位是秒。当值为0时，这意味着默认数据永不会被清除，意味着缓冲区有可能被旧的缓存条目填满，从而导致无法缓存新条目。

关于APC的配数参数还有一些，由于不常用不再尽述，详细说明请参考官网。

APC有一个做得很值得称道的地方，就是它附带了一个用于缓存管理的PHP脚本，让我们了解必要的缓存数据，包括缓存命中与未命中（忽略缓存时，表示内存已满），所有的用户空间缓存数据列表选项可查看或删除单个或所有项，等等。

我们在http://pecl.php.net下载apc源码包中有个apc.php文件，然后做一些小的修改，过程如下：

先找到defaults('USE_AUTHENTICATION',1);这行，如果你要看到详细的报告就将1改为0，这样不必登录就可以看到详细的APC内容。

下述两点是USER_AUTHENTICATION参数设置为1时，需要登录查看的情况：

（1）如果未登录，apc.php的"System Cache Entries"会把文件路径表示为"<hidden>"。

（2）需要增加预设的账号，例如：

```
defaults('ADMIN_USERNAME','admin');,
defaults('ADMIN_PASSWORD','password');
```

这里，账号和密码请修改为自己需要的。

APC管理界面如图4-9所示，其功能主要有以下几项：

- refresh data（数据刷新）。
- view host stats（查看主机状态）。

- system cache entries（系统缓存节点）。
- user cache entries（用户缓存节点）。
- version check（版本检查）。

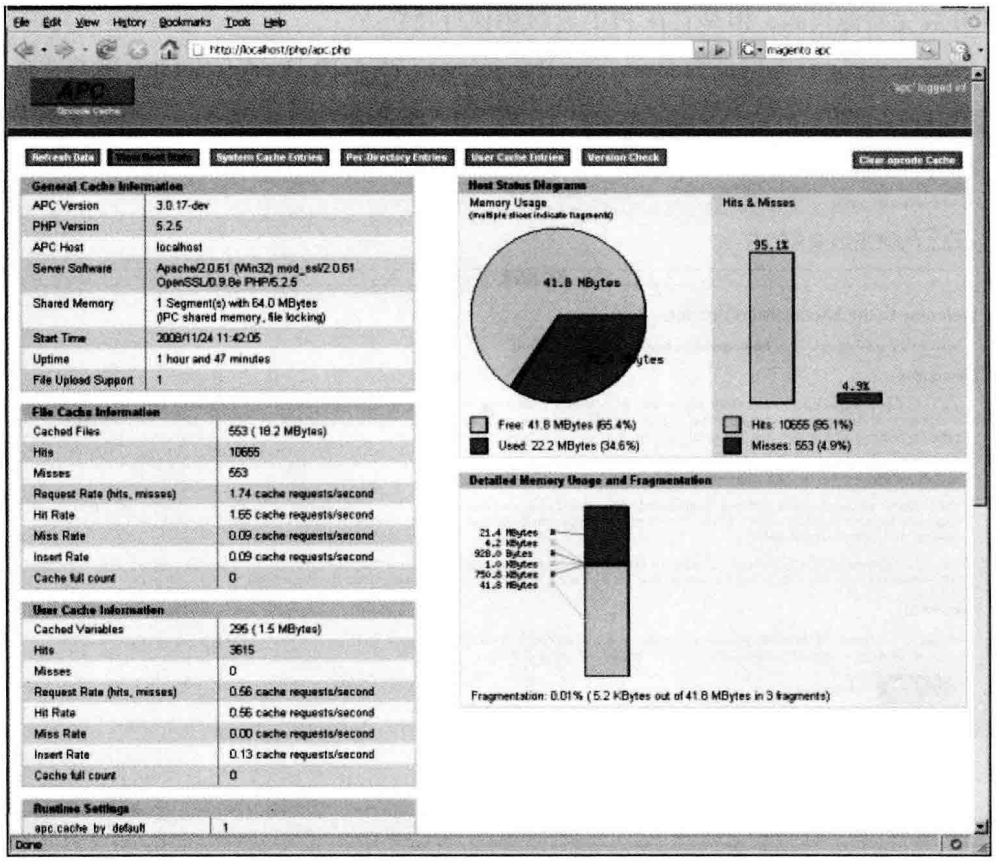


图 4-9　APC 的管理页面

APC 与 Zend OpCache 的优点如下：

（1）APC 有数据缓存 API，而 Zend OpCache 没有。

（2）APC 能够回收旧的无效的脚本占用的内存。APC 有内存管理器，可以将那些不再使用的脚本关联的内存进行回收。而 Zend OpCache 不同，它将这样的内存标记为“脏的”，但并不会回收它们。一旦“脏的”内存占用配置阈值的百分比达到一定值，Zend OpCache 就将自己重新启动。这种行为在稳定性上既有优势也有劣势。

4.6.2 eAccelerator 的安装配置

eAccelerator 是另外一个流行的 OpCode 中间代码缓存扩展，它是从一个叫 Turck-MMCache 的开源项目基础上发展而来的。eAccelerator 使用 GNU 通用许可证，和 APC 一样开放完整源代码，但是它在 PHP 社区中保持独立。

eAccelerator 官方网址为 http://eaccelerator.net。图 4-10 是 eAccelerator 网站首页。

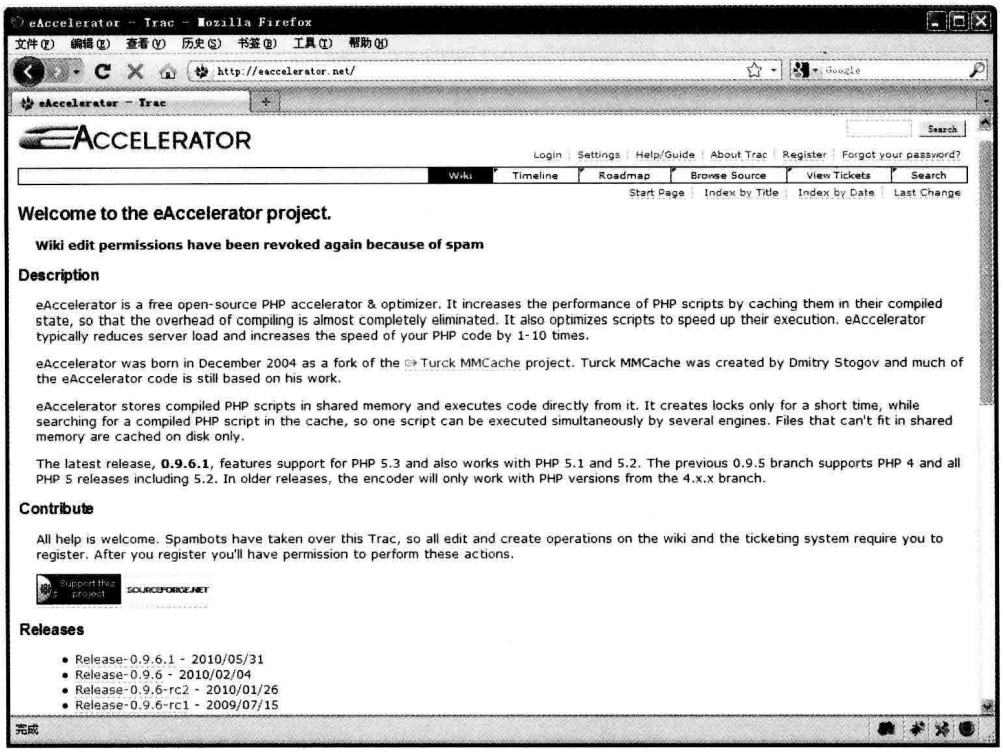


图 4-10 eAccelerator 网站首页

值得一提的是，在写本节时跨越了一年，eAccelerator 可以和 Zend Optimizer 同时存在于系统中，后来出现的 Zend OpCache 未经测试。

也许 eAccelerator 与 APC 的这点不同，兼顾了一些开发者需要数据缓存的需要。

在 Linux 系统中，安装和配置 eAccelerator 很简便，我们使用如下步骤：

```
#phpize ./configure --with-eaccelerator-shared-memory
#make
#make install
```

这时，将 eAccelerator 安装到 PHP 目录中，屏幕会显示 eaccelerator.so 所在路径，

例如：

```
installing shared extensions:
/usr/local/lib/php/extensions/no-debug-non-zts-20090626/
```

我们记住这个路径，在 php.ini 中加入该路径。如果存在 /etc/php.d 目录，需要复制 eaccelerator.ini 到该目录下，然后修改默认值。如果不存在，则修改 php.ini，加入以下内容：

```
[eaccelerator]
extension="/usr/local/lib/php/extensions/no-debug-zts-20090626/eaccelerator.so"
eaccelerator.shm_size="128"
eaccelerator.cache_dir="/tmp/cache/eaccelerator"
eaccelerator.enable="1"
eaccelerator.optimizer="1"
eaccelerator.check_mtime="1"
eaccelerator.debug="0"
eaccelerator.filter=""
eaccelerator.shm_max="0"
eaccelerator.shm_ttl="0"
eaccelerator.shm_prune_period="0"
eaccelerator.shm_only="0"
eaccelerator.compress="1"
eaccelerator.compress_level="9"
```

这个 eaccelerator.cache_dir 参数须和安装时创建的目录相同。所有参数中的“ shm”指共享内存，通过设定生命周期和共享内存的大小防止内存占用过大或溢出。

类似于 APC，eAccelerator 也不会自动删除缓存，除非用户指定某些数据删除，这意味着缓存数据会很快膨胀，有可能因此把速度拖得更慢。为此我们必须用 eaccelerator_ttl 参数来设置缓存的生命周期，以及缓存间隔多长时间会更新。和 APC 不同的是，eAccelerator 没有用户缓存的概念，所以生命周期会影响到所有的 Opcode 中间代码。

下面我们创建缓存目录，步骤如下：

```
#mkdir -p /tmp/cache/eaccelerator
#chmod 0777 /tmp/cache/eaccelerator
```

然后我们重新启动 Ngnix 或 Apache 服务器：

```
#service nginx restart
```

启动完成后，我们再查看 phpinfo.php，就可以看到 eAccelerator 的信息，如图 4-11 所示。

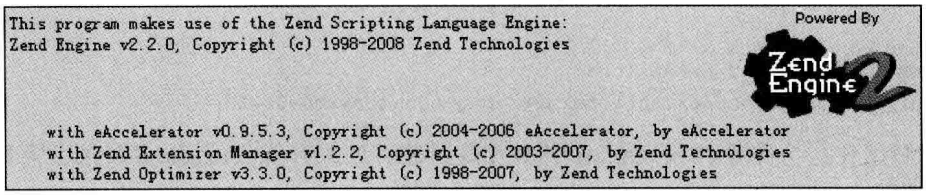


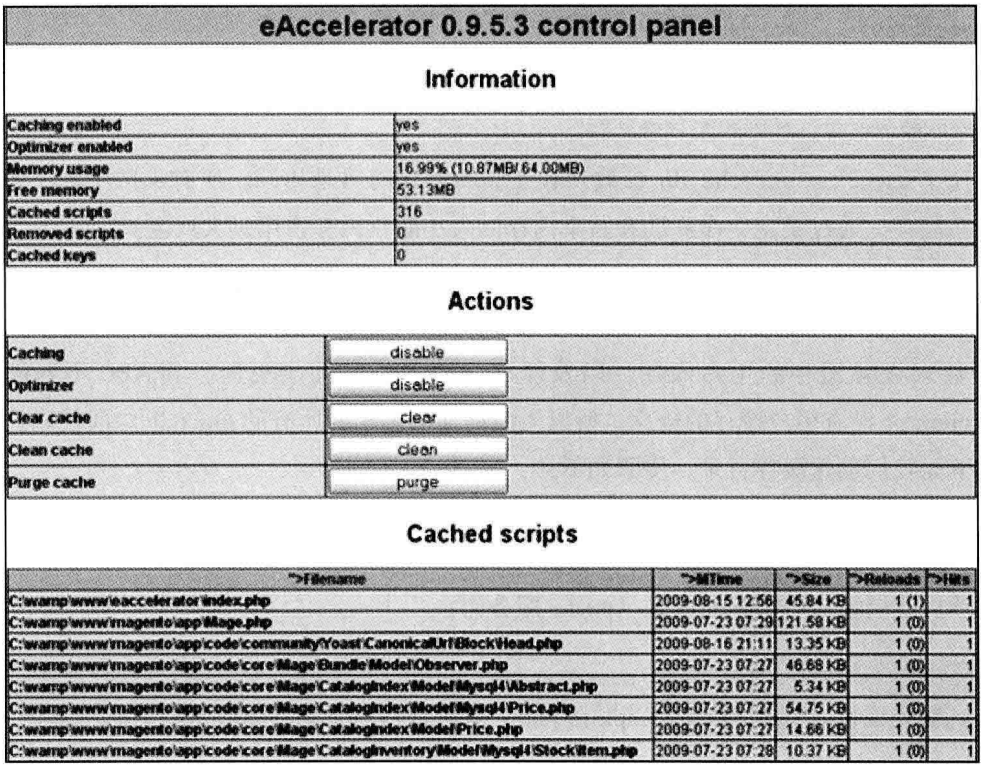
图 4-11 安装成功后的 eAccelerator 画面

和 APC 一样，eAccelerator 也附带了一个管理脚本，如果要使用它，需要在前面的参数中加入一行：

```
eaccelerator.allowed_admin_path = "/web/21cto.com/eaccelerator"
```

这样需要我们把 eAccelerator 的源码包里的 control.php 文件复制到 /web/21cto.com/eaccelerator 下，使它能够通过浏览器访问。

我们打开 control.php 时，输入默认用户名 admin 和密码 eAccelerator，会显示如图 4-12 所示的内容。



eAccelerator 0.9.5.3 control panel

Information

Caching enabled	yes
Optimizer enabled	yes
Memory usage	16.99% (10.87MB/ 64.00MB)
Free memory	53.13MB
Cached scripts	316
Removed scripts	0
Cached keys	0

Actions

Caching	disable
Optimizer	disable
Clear cache	clear
Clean cache	clean
Purge cache	purge

Cached scripts

">Filename	">MTime	">Size	">Reloads	">Hits
C:\wamp\www\eaccelerator\index.php	2009-08-15 12:56	45.84 KB	1 (1)	1
C:\wamp\www\magento\app\Mage.php	2009-07-23 07:29	121.58 KB	1 (0)	1
C:\wamp\www\magento\app\code\community\Yoast\CanonicalUrl\Block\Head.php	2009-08-16 21:11	13.35 KB	1 (0)	1
C:\wamp\www\magento\app\code\core\Mage\Bundle\Model\Observer.php	2009-07-23 07:27	46.68 KB	1 (0)	1
C:\wamp\www\magento\app\code\core\Mage\CatalogIndex\Model\Mysql4\Abstract.php	2009-07-23 07:27	5.34 KB	1 (0)	1
C:\wamp\www\magento\app\code\core\Mage\CatalogIndex\Model\Mysql4\Price.php	2009-07-23 07:27	54.75 KB	1 (0)	1
C:\wamp\www\magento\app\code\core\Mage\CatalogIndex\Model\Price.php	2009-07-23 07:27	14.66 KB	1 (0)	1
C:\wamp\www\magento\app\code\core\Mage\CatalogInventory\Model\Mysql4\Stock\Item.php	2009-07-23 07:28	10.37 KB	1 (0)	1

图 4-12 eAccelerator 的管理界面

这个用户名和密码可以在 control.php 文件里修改。这个控制面板有两大主要功能，一是控制 eAccelerator 的功能，如禁用或者启用 eAccelerator 的缓存和优化作用，删除缓存的内容，等等；二是查看已经缓存过的文件。

4.6.3　XCache 的安装配置

XCache 是由中国人 Mo0 开发的 OpCode 缓存工具，它原属于 LightHttpd Web 服务器下的子项目之一（http://xcache.lighttpd.net/）。

XCache 虽然在推出时间上晚于 APC 和 eAccelerator，但在性能方面表现非常优秀，目前在测评方面，XCache 要优于 APC 和 eAccelerator，而且社区活跃，版本的更新很快，最让开发者贴心的是，它的兼容性也很好，可以与 Zend Optimizer 以及 ionCube PHP Loader 并存。

XCache 支持 Liunx 和 FreeBSD 平台，但是目前尚未支持 Windows。

在 Linux 系统下安装 XCache

首先下载 XCache，本例中使用的是 1.3 版本，先用 cd 命令定位到一个目录如 src，再使用 wget 进行下载，然后解压缩与安装，系列命令如下：

```
#wget http://xcache.lighttpd.net/pub/Releases/1.2.2/xcache-1.3.0.tar.gz
#tar -zxf xcache-1.3.0.tar.gz
#cd xcache
#phpize
#./configure --enable-xcache
#make
#make install
```

安装完成后，记录下 XCache 提示的安装目录。然后编辑 php.ini 文件，加入 Xcache，作为 Zend 扩展。代码如下：

```
[xcache-common]
;; install as zend extension (recommended), normally "$extension_dir/xcache.so"
zend_extension = xcache.so
;; or install as extension, make sure your extension_dir setting is correct
; extension = xcache.so

[xcache.admin]
xcache.admin.auth = On
xcache.admin.user = "admin"
; xcache.admin.pass = md5($your_password)
xcache.admin.pass = "21232f297a57a5a743894a0e4a801fc3"
```

```
[xcache]
xcache.shm_scheme = "mmap"
xcache.size  = 32M
xcache.count = 1
xcache.slots = 8K
xcache.ttl   = 3600
xcache.gc_interval = 300

; Same as aboves but for variable cache
; If you don't know for sure that you need this, you probably don't
xcache.var_size  = 0M
xcache.var_count = 1
xcache.var_slots = 8K
xcache.var_ttl   = 0
xcache.var_maxttl   = 0
xcache.var_gc_interval = 300

; N/A for /dev/zero
xcache.readonly_protection = Off
xcache.mmap_path = "/dev/zero"
xcache.cacher =  On
xcache.stat   =  On
```

重启 Ngnix 或 Apache 服务器，然后在 phpinfo 中会看到 XCache 的提示，如图 4-13 所示。

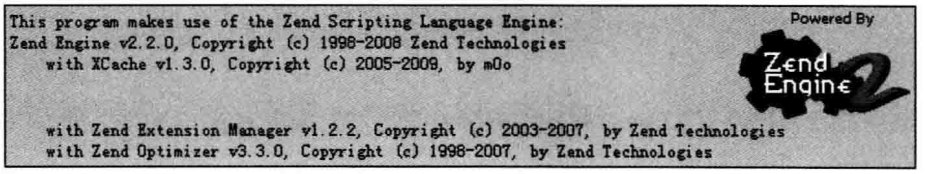


图 4-13　安装成功的 XCache 画面

注意，以上 XCache 的管理员密码均为 admin 的 md5 哈希值，请酌情修改为自己的密码。

4.6.4　使用 XCache 缓存

我们已经了解了 XCache 的安装和配置，下面我们使用一个例子来进行说明。首先，我们创建一个新闻表，用来保存新闻文章，结构如下：

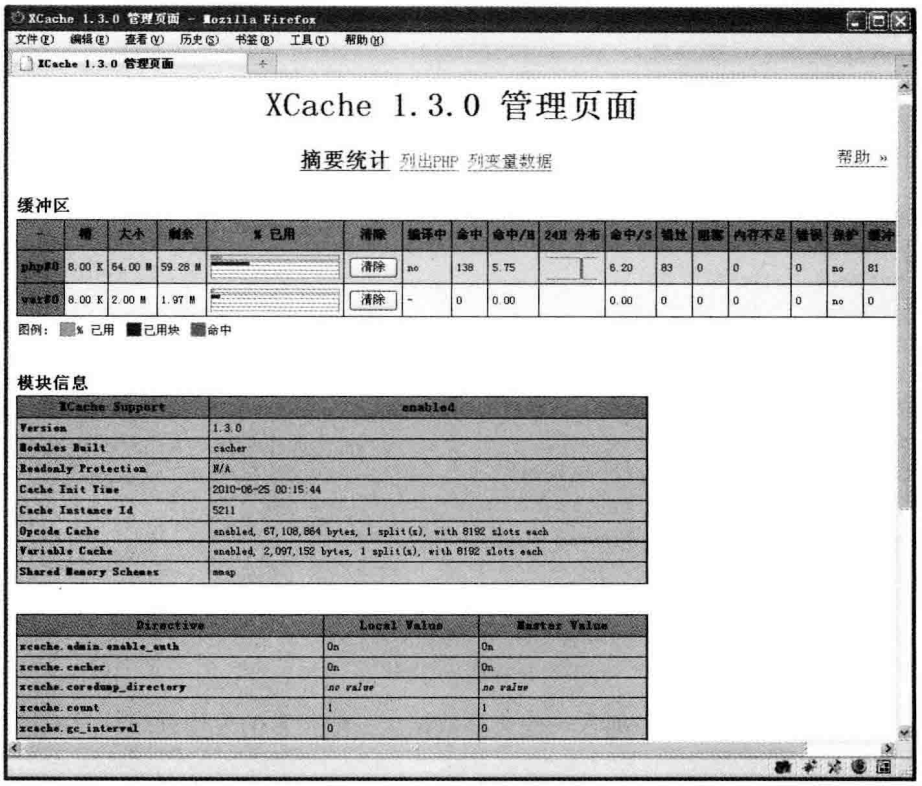


图 4-14　X XCache 的管理界面

```
CREATE TABLE IF NOT EXISTS `news` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `title` varchar(255) NOT NULL DEFAULT '',
  `alias` varchar(255) NOT NULL DEFAULT '',
  `title_alias` varchar(255) NOT NULL DEFAULT '',
  `introtext` mediumtext NOT NULL,
  `fulltext` mediumtext NOT NULL,
  `state` tinyint(3) NOT NULL DEFAULT '0',
  `sectionid` int(11) unsigned NOT NULL DEFAULT '0',
  `mask` int(11) unsigned NOT NULL DEFAULT '0',
  `catid` int(11) unsigned NOT NULL DEFAULT '0',
  `created` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `created_by` int(11) unsigned NOT NULL DEFAULT '0',
  `created_by_alias` varchar(255) NOT NULL DEFAULT '',
  `modified` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `modified_by` int(11) unsigned NOT NULL DEFAULT '0',
  `checked_out` int(11) unsigned NOT NULL DEFAULT '0',
  `checked_out_time` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
```

```
  `publish_up` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `publish_down` datetime NOT NULL DEFAULT '0000-00-00 00:00:00',
  `images` text NOT NULL,
  `urls` text NOT NULL,
  `attribs` text NOT NULL,
  `version` int(11) unsigned NOT NULL DEFAULT '1',
  `parentid` int(11) unsigned NOT NULL DEFAULT '0',
  `ordering` int(11) NOT NULL DEFAULT '0',
  `metakey` text NOT NULL,
  `metadesc` text NOT NULL,
  `access` int(11) unsigned NOT NULL DEFAULT '0',
  `hits` int(11) unsigned NOT NULL DEFAULT '0',
  `metadata` text NOT NULL,
  PRIMARY KEY (`id`),
  KEY `idx_section` (`sectionid`),
  KEY `idx_access` (`access`),
  KEY `idx_checkout` (`checked_out`),
  KEY `idx_state` (`state`),
  KEY `idx_catid` (`catid`),
  KEY `idx_createdby` (`created_by`)
) ENGINE=MyISAM  DEFAULT CHARSET=utf8;
```

另外，我们需要在该表中加入一些文章数据。一般正常的应用程序开发应该类似于这种代码：

```
function getNews()
{
    $GLOBALS['db-read']->query('SET NAMES utf8');
$query_string = 'SELECT * FROM news ORDER BY publish_up LIMIT 10';
    $result = $GLOBALS['db-read']->getAll($query_string);
    return $result;
}
```

下面我们使用XCache保存数据定义一个接口类：

```
interface ICacheManager
{
    public function fetch($key);

    public function store($key, $value, $ttl);

    public function delete($key);
}
```

这个接口有3个方法，分别为获取、保存和删除，它们的功能未实现，需要子类继

承后实现。

```
class XCacheCacheManager implements ICacheManager{
    public $default_ttl;
    static $_instance;

    public static function getInstance(){
            if(!(self::$_instance instanceof self)){
                    self::$_instance = new self();
            }

            return self::$_instance;
    }

    private function __construct(){
            //默认为10分钟
            $this->default_ttl = 600;
    }

    public function fetch($key){
            return xcache_get($key);
    }

    public function store($key, $value, $ttl){
            return xcache_set($key, $value, $ttl);
    }

    public function delete($key){
            return xcache_unset($key);
    }
}
```

其中 $ttl 为缓存的生存周期，我们在其他脚本中就可以引用该类进行缓存的更新和保存了。下面我们再改写前面的 getNews() 函数，代码如下：

```
function getNews(){
    $cm = XCacheCacheManager::getInstance();

    $query_string = 'SELECT * FROM news ORDER BY publish_up LIMIT 10';

    //是否已经缓存
    if($data = $cm->fetch(md5($query_string))){
        $result = unserialize($data);
    }
    else{
```

```
        //数据库为utf8编码
        $GLOBALS['db-read']->query('SET NAMES utf8');
        result = $GLOBALS['db-read']->getAll($query_string);

        $cm->store(md5($query_string), serialize($result), 3600);
    }

    return $result;
}
```

代码是不是很整洁？我们在保存缓存时使用了 MD5 的哈希串，用它来标识这一个特定的缓存数据集合，而且在后面也不用额外记住这个关键字。

我们想什么时候缓存到期更新，就设置 $ttl 的值，一旦生存周期到期，该缓存将会被刷新。这意味着，当网站被用户访问时，只有一个用户访问的是从数据库提取的内容，如我们的新闻数据库，而其他用户看到的将是缓存的结果，无论访问量多大，都不会产生数据库“瓶颈”。

如果我们需要对缓存更新，无论过期与否，只需添加一些代码到负责更新的函数中即可。这里我们有一个更新的函数 updateNews()，通过它来了解如何更新缓存和数据库的：

```
function updateNews($newsData,$newsId)  {
    $cm = XCacheCacheManager::getInstance();
    $query_string = 'UPDATE news SET fulltext='.$newsData. ' WHERE id='.$newsId.'';
        //此处为新闻更新语句
   $GLOBALS['db-read']->query($query_string);

     // 更新缓存
     $query_string = 'SELECT * FROM news ORDER BY publish_up LIMIT 10';
     $result = $GLOBALS['db-read']->getAll($query_string);

     $cm->store(md5($query_string), serialize($result), 3600);
}
```

该函数接收表单传递过来的新闻内容和新闻 id，更新完数据库后，再次更新高速缓存。

下面我们开始调用 getNews 函数显示数据库或 XCache 缓存中的内容，代码如下：

```
// 脚本开始执行时间
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
```

```
$starttime = $mtime;

// 本例为测试，请使用SPL auto-loader方式
include('ICacheManager.inc.php');
include('XCacheCacheManager.inc.php');
include('database.inc.php');
include('functions.inc.php');
// 创建一个数据库链接
$GLOBALS['db-read'] = new DbManager('db-read');

// 取得新闻
$news = getNews();

// 输出新闻内容
echo '<h1>PHP新闻联播</h1>';

foreach($news as $item){
    echo '<a href="' . $item->link . '" target="_blank">' . $item->title . '</a>
        <br />';
}

echo '<hr />';

// 显示脚本执行完毕后所需要的时间
$mtime = microtime();
$mtime = explode(" ",$mtime);
$mtime = $mtime[1] + $mtime[0];
$endtime = $mtime;
$totaltime = ($endtime - $starttime);
echo "页面执行时间共".$totaltime." 秒";
```

使用 XCache 显示的新闻如图 4-15 所示。

当第一次执行这个脚本时，执行时间会在 0.005 s 左右，再次刷新时就变成 0.001 s 左右，这证明第 2 次的内容已经是缓存内容了，我们可以感受 XCache 真正的快感。这样服务器就能承受一个高负载应用。

本节的例子虽然使用的是 XCache，但是它的函数与 APC、eAccelerator 的函数都很类似，如果你用的是这两种的其中一个，修改类中的函数即可。

4.6.5　APC、eAccelerator 和 XCache 三者的比较

APC、eAccelerator 和 XCache 都是非常有用也是非常类似的工具，每个都提供缓存性能优化和管理的功能。APC 和 XCache 是开放源代码的项目，用户可以在源代码上进

图 4-15 使用 XCache 显示新闻

行二次开发，开发自定义的函数。eAccelerator 的开源规则相对严格，但它是一个相对活跃的项目。

在真正用于生产环境时，需要团队确定哪些缓存工具合适，开发方面容易，并且性能提高很大。

前面提到过其他几个 Opcode 缓存解决方案，如 ionCube PHP accelerator 及 Zend Platform 等，它们要么不常见，要么不开源，因此本书选择介绍这 3 个主流的开源缓存解决方案，没有介绍闭源的解决方案。

尽管使用 OpCode 缓存技术提高性能是不可否认的，但如何适配在应用上还需要开发者自己来确定。

值得一提的是，本地缓存不应该变化太频繁，因为要不断地删除或修改缓存内容，在这种情况下磁盘的 IO 成本更高，因此缓存在此时是无用甚至是浪费。

另外，如果系统架构是多个 Web 服务器，要涉及用户缓存的分布和同步管理，多个服务器间要产生多个缓存副本，这点本地缓存的缺点就暴露出来了。

在复杂的应用程序需要做缓存分布式，以优化资源配置。这就是需要下面要讲述的 memcached 功能来发挥作用了。

目前已知的问题：

如果安装了 Zend Optimizer，它可以和 eAccelerator 配合工作，会和 APC 产生冲突。

APC 和 eAccelerator、XCache 在做几乎同样的事情，只是在某些方面，各有所长。如果加载多个肯定会引发讲点。因此两个选其一即可。

可见如果要求性能，我们首选 pecl-APC，兼容性和性能表现都非常优秀，但是不能与 iconcube、Zend Optimizer 等共存。如果需要 Zend Optimizer，就安装 eAccelerator 或 XCache。

若我们使用的是 PHP 5.5，则只需打开 Zend OpCache 选项即可。除非有特别的偏多、或特殊需求。

4.6.6　用户级别缓存

用户缓存是针对于应用程序层的一个概念，应用程序可以指定哪些缓存需要保存，哪些需要清除和覆盖，和我们前面讲过的数据级别缓存相类似。APC 支持用户缓存，而 eAccelerator 和 XCache 在新版本或有此功能。

用户缓存可以存储任意值，在应用程序可以定义为任意的 key 标识，但应确保 key 是独一无二的。另外，可以对该 key 的缓存设置一个生命周期时间。因为网站还是会定期更新的，因此需要定义一个时间周期去更新一次缓存。

确定最佳的缓存生存周期，是应用程序开发人员的责任。当周期结束时，系统将向数据库取得最新的数据后关闭，如果未到生存周期，则避免去查询 MySQL 数据库和计算的开销，这样则大大提高了应用程序的性能。

保存和取得缓存的方法是最常见的，APC 保存缓存的方法格式：

```
apc_store("key", $object, 3600 );
$stored_object = apc_fetch("key");
```

eAccelerator 保存缓存的方法格式：

```
eaccelerator_put("key", serialize($object), 3600 );
$store_object = unserialize( eaccelerator_get("key"));
```

对于 XCache，使用如下方法格式：

```
xcache_set("key", $data, 3600);
$store_object = xcache _get("key");
```

这 3 种工具的方法均使用同一顺序的参数：(key，value)，然后是缓存的生存周期，每个缓存的关键字是单词“ key”（当然是这个演示，在正式环境中使用是不好的命名），它们生存周期均为 1 小时。

eAccelerator 在保存缓存方面和另外两个工具有所区别，它不会自动序列化缓存数据。如果忘了序列化对象，eAccelerator 会报致命的错误，当应用程序试图检索数据时，会根本取不到结果，导致调试困难，这一点请各位开发者注意。

另外，有几种情况无须进行序列化，如布尔值和数据，这些是基本数据类型。另外，也不能在缓存中存储资源（Resource）数据类型，如 MySQL 连接句柄和文件句柄，这些必须是在每次执行时重新打开。最常见的缓存是存储一些消耗资源较高的内容，如数据库保存的内容、文档等，缓存会根据频率从内存中直接存取，从而达到性能的最大化。

4.7 使用 deflate 压缩页面

前面我们花更多的时间来优化 PHP 内部的缓存。另外，向客户端发送 HTML 之前还可增加一步优化，那就是压缩功能。HTTP 协议允许压缩传输数据，压缩可使 Web 应用提速并节省带宽。

关于页面压缩，我们通过图 4-16 来查看它所处的位置。

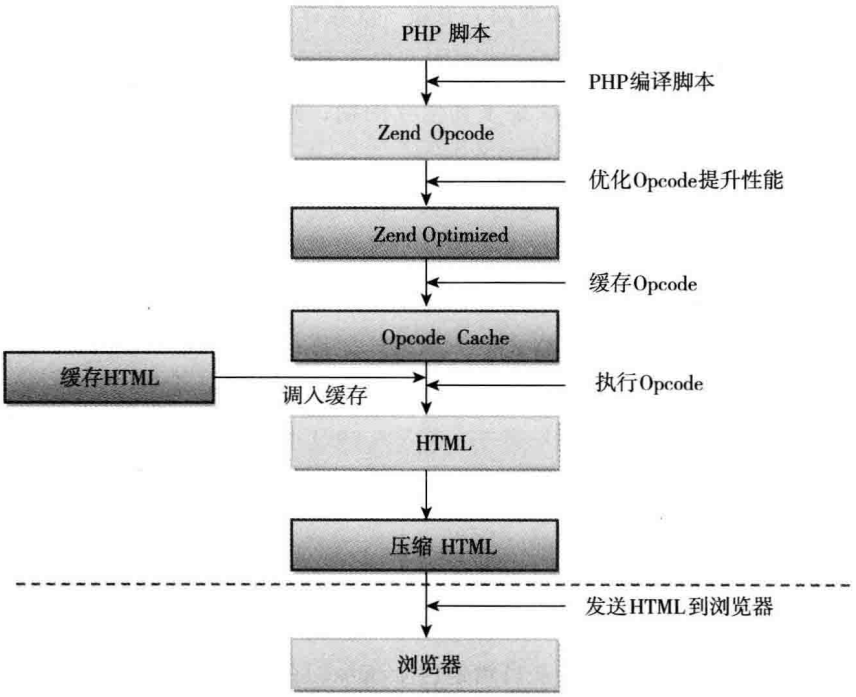


图 4-16 页面压缩位置示意图

在 Apache 中有一个标准压缩扩展模块可用，叫作 deflate（以前称为 gzip），可以让我们快速、轻松地压缩文件后，再发送给客户端浏览器。

首先安装时要确定已经安装 deflate 模块，然后在 htttpd.conf 中加入一行，如下：

```
LoadModule deflate_module modules/mod_deflate.so
```

然后再加入以下代码段：

```
<Location />
    AddOutputFilterByType DEFLATE text/html text/plain text/xml text/x-js text/
        css
</Location>
```

这表示会对 html、xml、js 和 CSS 文件进行压缩后再输出到浏览器，你可以根据自己的需求增加或删除不同的文件 MIME 类型。接下来请重启 Apache 服务器，使用 firefox 的 firebug 工具查看页面头的 Content-Encoding 一行，如果是 gzip，则内容已经成功被压缩。

4.8　内存数据库

内存数据库，即使用系统上的可用内存来保存数据。基于文件的缓存和基于内存的数据库系统有两个本质区别：首先，基于内存的缓存需要一个中间件，即要访问系统内存；其次，基于内存的缓存容易丢失，如停电后内存数据会全部丢失，被重载（通过 Web 服务器被重新启动），等等 。

基于内存的缓存提供了优于其他缓存的一些特点：最明显的，它们通常非常快（因为内存非常快），从而提高性能。马上要讲到的 memcached 充分表现了这种优势。

此外，基于内存的缓存通常不会出现锁定，而基于文件的缓存有此问题或者有权限属性等问题。

基于内存的缓存的缺点是，它是有限的内存系统。与基于文件的缓存（仅受限于硬盘空间）不是一个量级，大多数机器有数百 GB 的硬盘空间，它们仅有一个几十 GB 的内存空间。因此当使用内存缓存时，开发者须考虑使用内存缓存的影响。

按存储介质，APC 与 memcached 均属于内存缓存。前面已经介绍过 APC，下面开始了解 memcached。

4.8.1　关于 memcached

memcached（http://memcached.org/）是以 Danga Interactive 的 Brad Fitzpatric 为首开发的一款内存数据库，它可以运行在 Unix、Linux、Windows 及 Mac 上。memcache

实现了一个可分布在多台机器的散列表。除了 LiveJournal，还有 Facebook、Yahoo!、Twitter、Wikipedia 等，国内的新浪网、人人网、开心网、豆瓣、搜狐及赶集等站点均在大规模应用。

前面我们提到过，大多数的 Web 应用都将数据保存到数据库中，应用服务器从中读取数据并在浏览器中显示。但随着数据量的增大、访问的集中，就会出现数据库的负担加重、数据库响应恶化、网站显示延迟等重大影响，成为众多网站的一个大“瓶颈”。

前面提到的网站每天的页面浏览量在 1000 万～2000 万，粗算一下，每秒钟会有超过 1 万次的写请求、10 万次以上的读请求，单单硬盘 IO Wait 就把所有的 MySQL 线程阻塞了。

memcached 提供了高性能的分布式内存缓存服务，通过其缓存数据库查询结果，可减少数据库访问次数，以提高动态 Web 应用的速度，提高可扩展性。

用 memcached 是为了解决数据库服务器的 IO 性能“瓶颈”，提高性能和负载能力。图 4-17 所示为一般情况下 memcached 的用途。

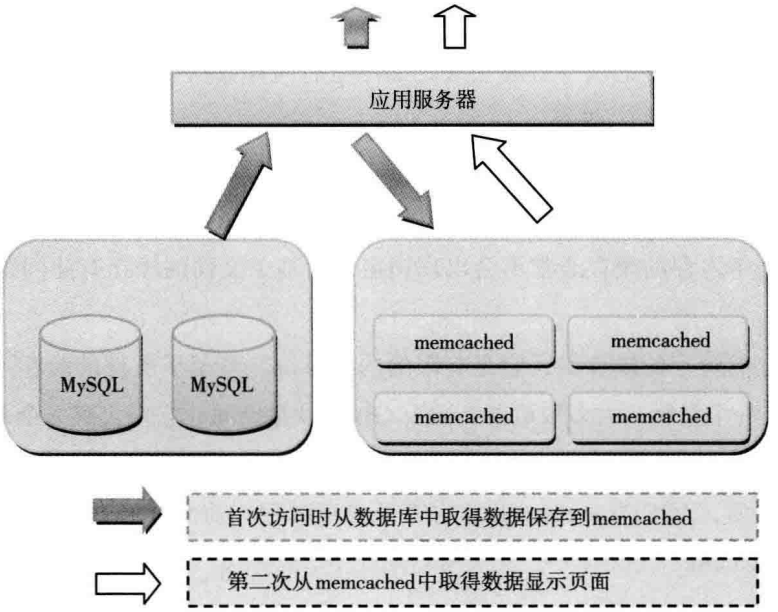


图 4-17　memcached 结构示意图

另外与 memcached 类似的项目还有 Ehcache（http://ehcache.org）、JBoss Cache、

Tokyotyant（http://1978th.net/tokyotyrant/）及 Redis（http://code.google.com/p/jredis/），这些产品也是分布式内存数据库系统，功能和协议与 memcached 大部分兼容，性能也具备竞争力。

4.8.2　memcached 架构

memcached 可能连接多台（几台或上百台）memcached 服务器组成一个内存池（memory pool），它使用系统未使用的内存，并能够很好地利用内存。

前面我们提到过，在单机的缓存系统，缓存分别保存在自己的服务器上，只能由该服务器上的应用程序使用，因此产生浪费。当我们试图把这些单机缓存合并时，需要缓存的一致性，这无疑是复杂的。图 4-18 描绘了单机缓存与 memcached 给架构带来的差异。

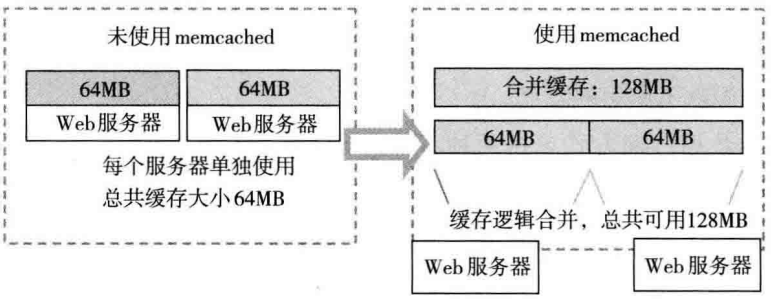


图 4-18　使用 memcached 与未使用 memcached 的集群对比

4.8.3　memcached 特性

memcached 服务器与客户端之间的通信协议没有使用像 XML 之类复杂的格式，而是用简单的基于文本的 ASCII 和二进制两种协议，这使得调试和客户端开发变得容易，也更透明。

memcached 是基于 libevent 库——一个非阻塞式的网络程序库开发的，因此能在 Linux、BSD、Solaris 等操作系统上能够发挥极高的性能。

libevent（http://monkey.org/~provos/libevent/）是一个事件触发的程序库，它将 Linux 的 epoll、BSD 类操作系统的 kqueue 等事件处理功能封装成统一的接口，即服务器的并发数量非常大，也能保持快速响应的能力。

为了提高性能，memcached 中保存的数据都存储在 memcached 自己管理的内存存储空间中，因为数据只存在于内存中，因此重启 memcached、重启操作系统都会导致全部

数据消失。

1. memcached 的内存存储处理

memcached 采用自己的内存存储处理，首先按照预先规定的大小，将内存分割成各种尺寸的块（chunk），并把尺寸相同的块分成组（chunk group），从而解决内存碎片问题。

memcached 根据收到的数据大小，选择最适合数据大小的 slab。memcached 中保存着 slab 内空闲 chunk 的列表，根据该列表选择 chunk，然后将数据缓存于其中。

由于分配都是固定长度的内存块，因此无法有效利用分配的内存空间。比如将 100 字节的数据组存放在能放 128 字节的 chunk 中，余下 28 字节是浪费的。

2. memcached 的数据处理算法

另外，memcached 本身是为缓存而设计的服务器，没有过多考虑数据的永久性问题，当内存容量达到指定值之后，就会基于 LRU（Least Recently Used，“删除最近最少使用”）算法自动删除不使用的缓存。

memached 数据过期方式包括两种：Lazy Expiration 和 LRU。memcached 内部不会监视记录是否已经过期，而是在进行 get 时查看记录的时间戳，检查记录是否过期。这种技术称为 Lazy（惰性）Expiration。因此，memcached 不会在过期监视上耗费 CPU 时间。

memcached 会优先使用已过期记录的内存空间，但也会发生追加新记录时内存不足的情况。此时 LRU 机制开始使用。因此，当 memcached 的内存空间不足时（无法从 slab class 获取新的空间时），就从最近未被使用的记录中搜索，并将其空间分配给新的记录。从缓存的实用角度来看，该模型是十分理想的。

3. memcached 客户端分布式结构与算法

尽管 memcached 是“分布式”缓存服务器，但服务器端并没有分布式功能。这好像很有意思，各个 memcached 不会互相通信以共享信息。那么，怎样进行分布式呢？这完全取决于客户端的实现，如图 4-19 所示。

无论使用 PHP 还是 Java、.NET，每种语言实现的客户端库都会包含至少一种分布算法来实现 memcached 分布式。

因此，笼统来说客户端库是通过一个分布算法和维护一个服务器列表来实现 memcached 分布式的，关于分布算法目前有两种选择：取模算法（modula hashing）和一致性

算法（consistent hashing）。

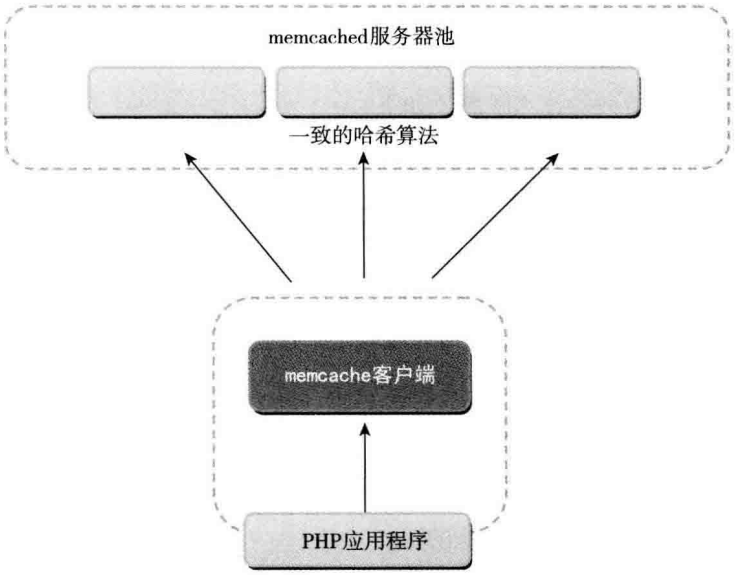


图 4-19　memcached 的访问框架

取模算法（modula hashing）是当前多数客户端库默认算法 [hash($key) % $svrNum]，就是根据服务器节点数的余数来进行分散，就是通过 hash() 函数求得的 key 的整数哈希值再除以服务器节点数并取余数来选择服务器。这种算法取余计算简单，分散效果好，但是缺点是，如果某一台机器宕机，那么应该落在该机器的请求就无法得到正确的处理，这时需要将宕掉的服务器从算法中去除，此时会有（N-1）/N 的服务器的缓存数据需要重新进行计算；如果新增一台机器，会有 N/（N+1）的服务器的缓存数据需要进行重新计算。对于系统而言，这通常是不可接受的颠簸（因为这意味着大量缓存的失效或者数据需要转移）。

一致性算法（consistent hashing）来源于 p2p 网络的路由算法，算法描述：hash 值一般为 unsigned int 型，因此对于 hash() 函数的结果应该均匀分布在 [0, 2^32-1] 间，把一个圆环用 2^32 个点来进行均匀切割，首先按照 hash() 函数算出服务器（节点）的哈希值，并将其分布到 0～2^32 的圆上。用同样的 hash() 函数求出需要存储数据的键的哈希值，并映射到圆上。然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器（节点）上。新增一个节点的时候，只有在圆环上新增节点逆时针方向的第一个节点的数据会受到影响；删除一个节点的时候，只有在圆环上原来删除节点顺时针方向的

第一个节点的数据会受到影响，因此通过 consistent hashing 很好地解决了负载均衡中由于新增节点、删除节点引起的 hash 值颠簸问题。

4. memcached 的技术限制

memcached 还包括一些技术限制，如下：

- 在 memcached 中可以保存的 item 数据量是没有限制的，只要内存足够。
- memcached 单进程最大使用内存为 2G，要使用更多内存，可以分多个端口开启多个 memcached 进程。
- 最大 30 天的数据过期时间，设置为永久的也会在这个时间过期，由常量 REALTIME_MAXDELTA 60*60*24*30 控制。
- 最大键长为 250 字节，大于该长度无法存储，由常量 KEY_MAX_LENGTH 250 控制。
- 单个 item 最大数据是 1MB，超过 1MB 数据不予存储，由常量 POWER_BLOCK 1048576 进行控制，它是默认的 slab 大小。
- 最大同时连接数是 200，通过 conn_init() 函数中的 freetotal 进行控制，最大软连接数是 1024，通过 settings.maxconns=1024 进行控制。
- 跟空间占用相关的参数：settings.factor=1.25, settings.chunk_size=48, 影响 slab 的数据占用和步进方式。

4.8.4 memcached 缓存策略

一般情况下，在应用程序中检索和存储数据，需要运行 SQL 语句查询或存取数据库，如果使用了缓存，可以修改程序先到缓存中检查是否有数据，如果没有再去数据库取。

我第一次使用 memcached 是在 A8.com 的项目中，因为在线音乐的请求数很大，比如每次播放都要都到 MySQL 曲库中查询，如果用户搜索了“张学友”的歌曲，那么第 1 次去曲库中查询，当第 2 次再搜索张学友时，将从 memcached 中取得，这样就减轻了数据库端的负载。

开发者可以为不同的数据类型采取不同的缓存策略，主要取决于数据改变的频率以及数据存储的类型。

表 4-2 是 memcache 的缓存类型。

表 4-2　memcache 的缓存类型

策　略	说　明
确定性缓存	客户端应用请求数据。如果数据已经在 memcached 中，应用程序直接返回数据；如果没有，则从 MySQL 数据库中取得并写入 memcached，然后再返回数据
非确定性缓存	如果数据一直在 memcached 中，这样对不经常更新的数据很有用。它可能要求保证数据一直驻留在 memcached 中
Session 缓存或状态缓存	保存用户 Session 会话数据，如登录状态、购物车等
主动缓存	这种缓存和非确定性缓存类似，当向 MySQL 写入数据时，缓存中的数据能够自动更新。可以在此处使用触发器及 MySQL 的 memcached 函数
文件系统或页面缓存	缓存的是模板或 HTML 代码
部分页面缓存	保存的是页面组件。如页面显示时，降低开销较大的查询来提供查询操作，比如评分或评论。可以定期生成这些组件保存在 memcached 中，页面显示时，从 memcached 中取出取可
缓存复制	将某台 memcached 的存储的数据，写入多个 memched 服务，确保数据完整性，允许有一定的冗余。

4.8.5　memcached 安装与配置

memcached 安装比较简便，本节会介绍几个主流系统的安装方法。

首先下载 memcached 服务器，本例描述的版本是 1.4.5，我们也可以直接从官方网站（http://memcached.org/）下载最新的版本。除此之外，memcached 用到了 libevent 库，需要先安装 libevent，本例中使用的版本是 libevent-1.4.13，官方下载地址：http://monkey.org/%7Eprovos/libevent/。

接下来我们分别将 libevent-1.4.13-stable.tar.gz 和 memcached-1.4.5.tar.gz 解包、编译，安装步骤如下：

```
#wget http://monkey.org/~provos/libevent-1.4.13-stable.tar.gz
#wget http://memcached.googlecode.com/files/memcached-1.4.4.tar.gz
#tar -xzf libevent-1.4.13-stable.tar.gz
#cd libevent-1.4.13-stable
#./configure --prefix=/usr
#make
#make install
#cd ..
#tar -xzf memcached-1.4.5.tar.gz
#cd memcached-1.4.5
#./configure --prefix=/usr
```

```
#make
#make install
```


提示 如果 libevent 不是安装在 /usr 目录下，那么需要把 libevent-1.2a.so.1 复制到 /usr/lib 目录中，否则 memcached 无法正常加载。

这样，memcached 已经安装启动完毕。我们可以使用如下指令显示 memcached 提供的参数和说明：

```
#/usr/local/bin/memcached -h
```

有关 memcached 启动的主要参数，见表 4-3：

表 4-3 memcached 命令参数表

命令参数	说 明
-p <num>	监听 TCP 端口号（port），默认为 11211
-d	以守护进程（deamon）方式运行 memcached
-u <username>	运行 memcached 的账户，非 root 用户
-m <num>	最大的内存使用，单位是 MB，缺省是 64 MB
-c <num>	最大软连接数量，缺省是 1024 MB
-v	输出警告和错误信息
-vv	打印客户端的请求和返回信息
-h	打印帮助（help）信息
-i	打印 memcached 和 libevent 的版权信息（infomation）

运行以下命令来启动 memcached 守护程序，运行 memcached 守护程序很简单，只需一个命令行即可，不需要修改任何配置文件。

以当前用户身份登录并以背景方式运行以下命令：

```
$/usr/local/bin/memcached &
```

也可以守护（deamon）进程的方式运行：

```
$/usr/local/bin/memcached -d
```

以日志的方式运行：

```
$/usr/local/bin/memcached -u root -vv >>/usr/logs/memcached.log 2>&1 &
```

其中，-vv 将显示错误警告信息和各客户端的命令和响应。

测试 memcached 启动是否成功，在 Linux 系统下可以使用如下指令：

```
#pa auxxww | grep memcached
```

在 Windows 系统下可以借助任务管理器来查看。我们也可以使用 telnet 工具来连接 memcached。命令如下：

```
#telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
stats
STAT pid 4402
STAT uptime 1032
STAT time 1231155683
STAT version 1.4.5
STAT pointer_size 32
set test1 0 0 4
test
STORED
get test1
VALUE test1 0 4
test
END
delete test1
DELETED
get test1
END
```

使用 telnet 测试 memcached，如图 4-20 所示。

有必要对刚才输入的命令做一下简要说明：

（1）我们使用 stats 显示当前 memcached 服务的参数和状态。

（2）Set 是将键值存储的验证。

（3）get 取得指定的键值。

（4）delete 是删除某个键值。

通过字面已可以很容易理解这些命令和意义。前面提到过，memcached 包括一个 ASCII 和二进制的客户端协议。上面的示例是使用 ASCII 协议（因此可以支持 Telnet 互动），这是一个非常简单的协议，详细的介绍请参见：http://github.com/memcached/memcached/blob/master/doc/protocol.txt.。

还有几个其他用于测试 memcached 的工具，如 memcached-tool，它的下载地址：http://code.sixapart.com/svn/memcached/trunk/server/scripts/memcached-tool，这是一个 perl 写的工具，下载之后，可以使用如下代码指定测试：

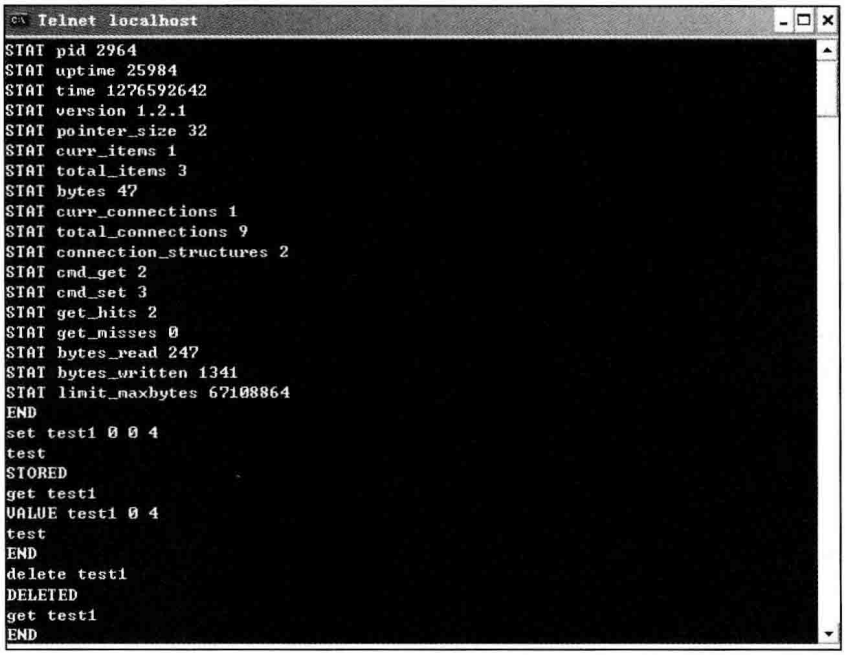


图 4-20 使用 telnet 测试 memcached

```
#perl memcache-tool 192.168.1.20:12111
```

另外，我们还可以使用 Livebookmark 图形界面监控 memcached 服务器。下载地址：http://livebookmark.net/memcachephp/memcachephp.zip。

解压缩后，是一个 memcache.php 文件，修改文件开始部分的 memcached 服务器设置代码：

```
$MEMCACHE_SERVERS[] = '192.168.1.20:12111';
$MEMCACHE_SERVERS[] = '192.168.1.21:12111';
```

将 memcache.php 部署到 Web 服务器上即可。

4.8.6 使用 memcached 做分布式 Session

在默认状态下，PHP 使用文件方式做 Session 存储，磁盘的 I/O 性能肯定没有内存中存取快，因此改用 memcache 来存储 Session 在读写速度上会快很多，而且在多台服务器集群时，使用 memcached 能够有效地解决 Session 共享的问题。架构确实变复杂了一些，但应用起来却极为简便，几乎不需要修改任何程序代码，通过修改几行配置信息即可实现。

我们配置好 memcached 服务器后，修改 php.ini 配置文件的代码如下：

```
session.save_handler = memcache
session.save_path = "tcp://127.0.0.1:11211"
```

也可使用在网站目录下的放置 Apache 的 .htaccess 文件，内容如下：

```
php_value session.save_handler "memcache
php_value session.save_path  "tcp://127.0.0.1:11211"
```

第一种方法在虚拟主机时可能无法做到，第二种方法可能会更方便，还有一种方法就是直接在 PHP 脚本中写入以下两行：

```
ini_set("session.save_handler", "memcache");
ini_set("session.save_path", "tcp://127.0.0.1:11211");?
```

如果在使用多个 memcached 服务器时用逗号隔开，还可以带额外的参数 persistent、weight、timeout、retry_interval 等，例如：

```
$session_save_path = "tcp://<memcache_server1>:11211?persistent=1&weight=1&tim
    eout=1&retry_interval=15, udp://<memcache_server2>:11211";
```

你可能看到在上面的例子中有用 tcp 的，也有用 udp 的，如果想使用 udp 支持，需要确保你的 memcached 服务器启用，也确保 Web 服务器连接到 memcache 服务器端口（多半在防火墙规则）正常。重新启动 Apache 后，将开始使用的 memcache 存储 PHP 的 Session。

启动 memcachede 服务后，我们在程序中使用 memcache 来做 Session 存储，PHP 脚本代码如下：

```
<?php
session_start();
if (!isset($_SESSION['memcached_test'])) {
    $_SESSION['memcached_test'] = time();
}
$_SESSION['add_test'] = time();
echo $_SESSION['memcached_test'];
echo "<br /><br />";
echo $_SESSION['addtest'];
echo "<br /><br />";
echo session_id();
```

我们看 Session 是否已经保存到 memcached，使用刚才显示的 sessionid 去 memcached 里查询。代码如下：

```
$memcache = memcache_connect('localhost', 11211);
var_dump($memcache->get('13762291213c65cedec65b0883238c278eeb573e077'));
```

执行后会有类似如下的输出：

```
string(37) "memcached_test|i:1177556731;add_test|i:1177556881;"
```

这表示 Session 已经保存，证明 Session 存取均已正常。

需要留心的是，如果使用 memcached 作为 Session 存储，要确保 memcached 的服务正常工作，否则 Session 相关功能将不起作用，这样 PHP 的处理就多了一层外面的依赖。因为 memcached 是使用内存的，这样当用户量比较大时，就可能由于内存方面原因导致 Session 时长上的问题，Session 的实际失效时长达不到设定的失效时长（由 memcached 在内存不够时的处理机制决定）。

4.8.7 两个 memcached 扩展

很有趣的是，在 PHP 的 pecl 站点，有两个实现功能类似的 memcached API 扩展（有时也被称为客户端），它们分别是 memcache（http://pecl.php .net/package/memcache）和 memcached（http://pecl.php.net/package/memcached）。

这两个扩展库和 memcached 的名字非常相近，而且功能于伯仲之间，有时确实会让人糊涂。

按时间追溯，memcache 在 2004 年 9 月就已经推出，在很多 Linux 系统中属于标准配置，也就是原生版本，而 memcached 扩展却是在 2009 年 1 月才发布。

显示 memcached 扩展比较新，它几乎支持 memcached 的所有特性（如 CAS、Delayed Get, Append/Prepend 等），但是它要依赖 libmemcached 才能运行。

所以如果我们不使用如 Delayed Get 这样的特性，又不想多依赖 libmemcached 库，完全可以使用 memcache 扩展，反之请选择 memcached 扩展。

另外，当发现键值有错误时，memcache 会尝试转换成一个合法的键值，而 memcached 在 set 或 get 操作时会直接返回布尔值 false，中止操作。鉴于性能和应用程度，本书中主要介绍 memcache 扩展的用法，如果使用 memcached 客户端的开发者也没关系，其中大部分方法是类似的。

4.8.8 安装 pecl::memcache 扩展

安装 memcache，我们从官方网站（http://pecl.php .net/package/memcache）下载。本

例中使用的版本是 2.25，安装步骤如下：

```
#tar -vxzf memcache-2.2.5.tgz
#cd memcache-2.2.5
#/usr/local/php/bin/phpize
#./configure --enable-memcache --with -php-config=/usr/local/php/bin/php-config
   --with -zlib-dir
#make
#make install
```

安装完后会有类似这样的提示：

```
Installing shared extensions: /usr/local/lib/php/extensions/no-debug-non-
    zts-20090626/
```

我们需要将它编译好的扩展库复制到 PHP 的扩展目录中：

```
#cp /usr/local/lib/php/extensions/no-debug-non-zts-20090626/memcache.so /usr/
     lib/php/modules/
```

提示　/usr/lib/php/extentions/ 是 php.ini 中默认的 extension_dir 所指，请读者修改自己服务器上的目录。

然后我们再打开 php.ini 配置文件，将下面的代码加到文件末尾：

```
extension=memcache.so
[Memcache]
memcache.allow_failover = On
memcache.max_failover_attempts = 20
memcache.chunk_size = 8192
memcache.default_port = 11211
;memcache.hash_strategy = "standard"
;memcache.hash_function = "crc32"
```

此时重启 Nginx 或 Apache 服务器，执行以下命令：

```
#php -i | grep memcache
```

如果一切顺利，或在浏览器中查看 phpinfo，会看到相关 memcache 的信息。如图 4-21 所示。

4.8.9　memcached 数据存取方法

这里为大家提供一个完整的 memcached 缓存类，可以处理基本的缓存操作。我们把常用的操作已经封装在方法中，开发者可以直接使用。代码如下：

memcache support	enabled
Version	3.0.2
Revision	$Revision: 1.83.2.28 $

Directive	Local Value	Master Value
memcache.allow_failover	1	1
memcache.chunk_size	32768	32768
memcache.default_port	11211	11211
memcache.hash_function	crc32	crc32
memcache.hash_strategy	consistent	consistent
memcache.max_failover_attempts	20	20
memcache.protocol	ascii	ascii
memcache.redundancy	1	1
memcache.session_redundancy	2	2

图 4-21 memcached 的状态信息

```
class Memcache_Cached {
   protected $memcacheObj;
   public function __construct($host, $port){
      $this->memcacheObj = new Memcache();
      $this->addConnection($host, $port);
   }
   public function addConnection($host, $port){
      $this->memcacheObj->connect($host, $port);
   }
   public function write($key, $data, $expires = 0, $flag = false) {
      return $this->memcacheObj->set($key, $data, $flag, $expires);
   }
   public function read($key, $flags = false) {
      return $this->memcacheObj->get($key, $flags);
   }
}
```

4.9 缓存的陷阱

缓存似乎是玫瑰，虽然有好处，能提高应用程序的性能，但如果不能很好地掌控它，也会让人感觉很棘手。

开发者有个很最重要的任务，要识别哪些功能需要用缓存，哪些不需要用缓存。缓存是附加在应用程序中的可选项，它不支持一个持久存储机制，最后还是要保存到数据

库。即使缓存依赖很重的应用程序，也要有缓存系统之外的解决方案。

开发者还需确保缓存的数据在应用程序的状态是当前和反映最好的，旧数据、脏数据、已失去实用性的数据应及时丢弃。

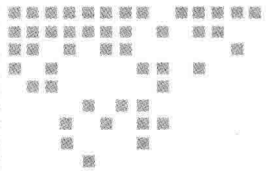
虽然有些项目旧数据的数量是可以接受的，但许多应用程序的显示数据则不能是虚假的或是过期的数据，如股票、球赛实时播报、聊天室等。规避这些缺陷也是一项不小的任务，一个精心策划的缓存系统，将有助于缓解或消除这些问题。

开发者要遵守以下准则，可以有效地避免缓存陷阱。例如：

- 标准的缓存接口 API。通过标准化的缓存 API，会有两个明显的优点：首先，外部开发者获得 API 是简单的方法，包括保存、获取和信息过期处理等；其次，开发者可以轻松切换后端，只要保持一致的 API，访问 API 的代码无须修改，可无缝过渡，也能更多控制在应用程序中的缓存行为。
- 建立规则保证数据真实。建立硬性和数据保真度的规则，确立什么样的过期数据是可以接受的。建立成文的规则，使每个开发者意识到与规则有关数据的真实度，降低过期或错误数据的使用。
- 在请求前，预先缓存好数据。使用 crontab 或其他方法把用户需要的数据预先缓存好，当用户请求时，可快速将缓存内容直接输出给用户。比如 Facebook 就使用了这种缓存，它们在实际场景中大量保存用户的行为和好友信息，为用户在检索其他好友时提供一个简单和快速的解决方法。

4.10　本章小结

缓存对大并高负载的网站是一个很好的解决方案，不仅提高了整体性能，同时减少了服务器负载。作为开发者需要认真规划好自己的缓存系统，实施有效的缓存技术是网站性能改善的重要技术指标之一。



Chapter 5

第 5 章

PHP 网络编程

进入移动互联网时代以后，后端的开发越来越重要。PHP 主要专注于 Web 后端开发，提供了很多的网络模型和 API，使得开发变得越来越方便和强大。

本章我们从 Socket 技术开始讨论，包括 Socket 协议、服务器端与客户端的编写。用 Socket 可以开发很多实用功能，如 SMTP 邮件处理、FTP 操作、HTTP 端 POST 提交、特殊报文通信、Whois 查询等。另一个部分内容是 cURL，包括 cURL 的原理详解，以及模拟用户请求、页面抓取、数据分析提取、模拟登录、管理 Cookie 等。

5.1 Socket 编程

5.1.1 Socket 原理

由于历史的原因，Socket 被译成“套接字”。这个词听起来有些抽象，我们可以将它理解成一个用来描述 IP 地址和端口的概念。Socket 不是一个程序，也不是协议，是由操作系统提供的通信层的一组 API。

Socket 位于 TCP/IP 协议的传输控制层，提供客户 / 服务器模式的异步通信，即客户向服务器发出服务请求，服务器接收到请求后，提供相应的反馈或服务，也就是两台主机之间通信的通道。应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样，如图 5-1 所示。

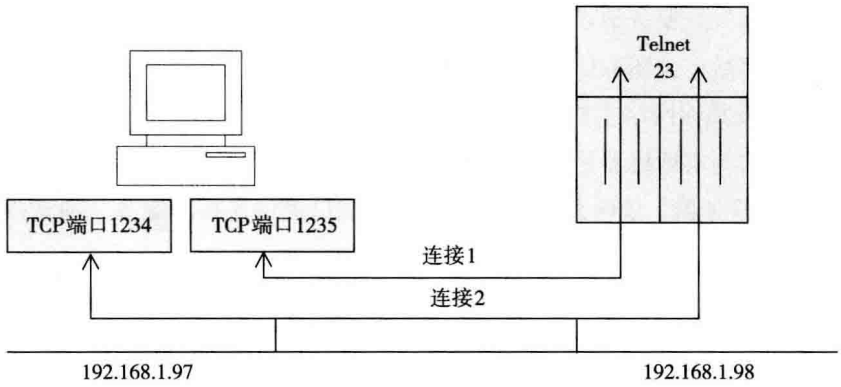


图 5-1　使用相同编号和唯一编号连接到设备

客户 / 服务器模式在操作过程中采取的是主动请求方式，服务器首先启动并呈伺服状态，收到客户端请求被动提供相应服务，如图 5-2 所示。

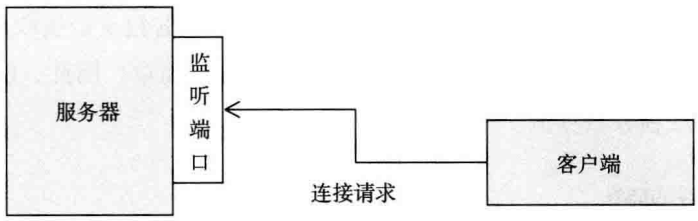


图 5-2　客户端发出连接请求

服务器处理步骤如下：

（1）打开一个通信通道并告知本地主机，它愿意在某一 IP 地址和端口上接收客户请求。

（2）等待客户请求到达该端口。

（3）接收到重复服务请求，处理该请求并发送应答信号。

接收到并发服务请求，要激活一个新进程来处理这个客户请求（如 Linux 中用 fork 和 exec）。新进程处理此客户请求，并不需要对其他请求做出应答。服务完成后，关闭此新进程与客户的通信链路，并终止服务。

（4）返回第（2）步，等待另一客户请求。

（5）关闭服务器。

客户端的处理步骤如下：

（1）打开一通信通道，并连接到服务器所在主机的特定端口。

（2）向服务器发送服务请求报文，等待并接收应答，继续提出请求。

（3）请求结束后，关闭通信通道并终止。

Socket 是连接其他网络主机的一种方法，该主机可以是互联网或局域网上的任何一台计算机，只要它与本地机器是连通的即可。

创建 Socket 服务器，监听外部请求并在指定端口提供连接的服务。通常作为一种服务或一个系统守护进程持续运行。

在 TCP/IP 协议族中的主要 Socket 类型为：流套接字（stream socket），传输层使用 TCP 协议，提供了一个可信赖的字节流服务；数据报套接字（datagram socket），传输层使用 UDP 协议，提供了一个“尽力而为”的数据报服务，最长一次可以发送 65500 个字节的数据。TCP 协议会确保没有数据丢失（如果数据包丢失，它将被重新发送），非常适合用于发送图片、文件或其他信息必须全部接收和反馈（如电子邮件）的服务。

UDP 称为用户数据报协议，这是一种无连接协议，如 TCP 它可以遵守 IP 协议。不同的是，UDP 提供很少的错误恢复服务，没有对另一侧数据包是否按收成功的确认，也没有告诉另一侧要按某种顺序按照的机制。这样的好处是简单。因此，UDP 特别适合于流数据，如音乐或视频数据流。

5.1.2 Socket 函数

PHP 提供给开发者的 Socket API 有两种，一种内置在 PHP 内核中，这些函数只能做主动连接而无法实现端口监听相关的功能，另一种是外部 PECL 扩展库，支持监听和交互模式。

我们首先来看一下集成于 PHP 内部的 Socket 函数。

内置 Socket 函数

PHP 内置的 Socket 函数包括 fsockopen 和 pfsockopen。它的格式如下：

```
Resource fsockopen (string $hostname[, int $port = -1 [, int &$errno [, string
    &$errstr [,float $timeout = ini_get("default_socket_timeout") ]]]] )
```

该函数的功能是初始化一个 Socket 套接字并连接到目标主机（hostname）。

psockopen() 函数实现的是持久化连接，也被称为长连接。你或许猜到，这个 p 就是 persistent 的缩写。psockopen 函数的格式如下：

```
resource pfsockopen ( string $hostname [,int $port = -1 [, int &$errno [, string
    &$errstr [,float $timeout = ini_get("default_socket_timeout") ]]]] )
```

它与 socketopen 函数功能相同，唯一区别就是建立的是长连接，即客户端与服务器端建立连接后不再断开，完成报文的发送与接收后再断开。

这两个函数执行后都会返回一个资源编号，这个资源几乎可以使用所有文件操作函数对其进行操作，如 fgets()、fwrite()、fclose() 等。比如 fread() 从文件指针读取指定长度的字节，该函数在读取完规定长度字节数，或到达文件尾 EOF，或（对于网络流）当一个包可用时就会停止读取文件，取决于先碰到哪种情况。来看以下脚本实例，如代码清单 5-1 所示：

代码清单5-1　Socket客户端

```
$fp = fsockopen("www.google.com", 80, $errno, $errstr, 30);
if (!$fp) {
    echo "$errstr ($errno)<br />n";
} else{
    $out = "GET / HTTP/1.1\n\n";
    //发送数据
    fwrite($fp, $out);
    //接收数据
    while (!feof($fp)) {
        echo fgets($fp, 128);
    }
    fclose($fp);
}
```

这个 fsockopen() 函数创建一个 Socket 连接，fwrite 函数负责发送数据，fread 接收数据，类似于从文件流中读 / 写。

上面是创建了一个基于 TCP 协议的 Socket 连接。要创建 UDP socket 连接，需像这样指定：

```
$fp = fsockopen("udp://127.0.0.1", 13, $errno, $errstr);
```

该流包装器可以换为 tcp://，我们可以概括一个通用的语法来概括 Socket 地址：

```
$protocol://$host:$port
```

5.1.3　PECL Socket 函数库

首先进行准备工作。如果你的 PHP 环境没有安装 Sokcet 扩展，可以使用如下方法：

```
./configure --enable-sockets make && make install
```

执行完毕后，它会自动修改 php.ini，只需要重启你的 Nginx 或 Apache 服务器。

如果是 Windows 系统，可以手工修改 php.ini 文件，打开 php_sockets.dll 即可。

完成以上步骤，意味着我们拥有了 PHP 强大的 Socket 功能，其中包括：监听端口、阻塞及非阻塞模式的切换、多客户端交互式处理，等等。

下面开始了解 php_sockets 这一扩展库提供的 Socket 函数族和应用实践。

1. socket_create

首先了解一个重要函数：socket_create()。

该函数用来创建一个 Socket。它有 3 个参数：一个表示协议、一个表示 Socket 类型、一个表示公共协议。如果 socket_create() 函数运行成功，返回一个包含 Socket 的资源类型，如果不成功，则返回 false。它的完整格式如下：

```
resourece socket_create(int protocol, int socketType, int commonProtocol);
```

socket_create 函数的参数：protocol 部分可以有如下值，为了描述和阅读方便，我们把说明汇总成表格，见表 5-1。

表 5-1 Socket 协议值和说明

名字 / 常量	描　述
AF_INET	这是大多数用来产生 Socket 的协议，使用 TCP 或 UDP 来传输，使用 IPv4 地址
AF_INET6	与以上类似，区别是使用 IPv6 地址
AF_UNIX	本地主机协议，用于 Unix 和 Linux 系统上，很少使用，一般为客户端和服务器在同一台的情况下使用

第二个参数是指定 Socket 通信的类型，它的值可以有 3 个常量，见表 5-2。

表 5-2 Socket 通信类型

类型（常量）	描　述
SOCK_STREAM	这个协议是一个按照顺序的、可靠的、数据完整的、基于字节流的连接。是使用最多的 Socket 类型，这个 Socket 使用 TCP 来进行传输
SOCK_DGRAM	这个协议是无连接的、固定长度的传输调用。该协议是不可靠的，使用 UDP 来进行它的连接。 SOCK_SEQPACKET 这个协议是双线路的、可靠的连接，发送固定长度的数据包进行传输，必须把这个包完整地接受才能进行读取
SOCK_RAW	这个 Socket 类型提供单一的网络访问，这个 Socket 类型使用 ICMP 公共协议（ping 和 traceroute 使用该协议）
SOCK_RDM	这个类型是很少使用的，在大部分的操作系统上没有实现，它提供给数据链路层使用，不保证数据包的顺序

第 3 个参数用来指定 Socket 使用哪种类型的处理协议，包括 ICMP、UDP、TCP，这 3 个协议见表 5-3。

表 5-3　Socket 处理协议类型

名字 / 常量	描　　述
ICMP	互联网控制消息协议，主要使用在网关和主机上，用来检查网络状况和报告错误信息
UDP	用户数据报文协议，它是一个无连接、不可靠的传输协议
TCP	传输控制协议，这是一个使用最多的、可靠的公共协议，它能保证数据包到达接受者那里，如果在传输过程中发生错误，那么它将重新发送出错数据包

2. socket_bind

该函数用于将 IP 地址与端口绑定到 socket_create 所创建的资源中。函数格式如下：

```
ool socket_bind ( resource $socket , string $address [, int $port = 0 ] )
```

可以看到 socket_bind 的 3 个参数，下面我们详细说明：

（1）Socket。用 socket_create() 函数创建的一个有效的套接字资源。

（2）address。如果套接字是 AF_INET，那么 address 必须是一个四点分法的 IPv4 地址（例如 127.0.0.1）；如果套接字是 AF_UNIX，那么 address 是 Unix/Linux 套接字的一部分（例如 /tmp/my.sock）。

（3）port（可选填）。参数 port 仅仅用于 AF_INET 套接字连接的时候，并且指定连接中需要监听的端口号。

3. socket_listen

在绑定 Socket 后，服务器端使用此函数监听客户端数据。函数格式如下：

```
bool socket_listen ( resource $socket [, int $backlog = 0 ] )
```

其中第一个参数是由 socket_create 创建的 socket 资源，第二个参数为可选项，表示允许的最大连接数。

4. socket_set_block

此函数设置为非阻塞模式。函数格式如下：

```
bool socket_set_block ( resource $socket )
```

当设置为非阻塞模式时，会立即返回。与其对应的就是阻塞模式，也就是没有完成任务不能返回，直到对方有反馈才能继续下一步处理。

当用户连接较多时，设置为非阻塞模式是必要的。试想如果设计为阻塞模式，有两个以上客户端连接，服务器在处理一个客户端的请求，另外一个客户端的请求就被阻塞，只有等前面的客户事情处理完，后面的请求才会被响应，效率低得可想而知。

5. socket_write

这个函数用于向 Socket 资源写入数据，格式如下：

```
int socket_write ( resource $socket , string $buffer [, int $length = 0 ] )
```

注意：此函数只是向 Socket 资源写数据，并没有执行发送（Send）操作。

6. socket_send

socket_send 函数用于向已连接的 Socket 资源发送数据，格式为：

```
int socket_send ( resource $socket , string $buffer , int $len , int $flags )
```

7. socket_read

socket_read 函数用来从 Socket 中读取指定长度的数据。格式如下：

```
string socket_read ( resource $socket , int $length [, int $type = PHP_BINARY_READ ] )
```

前两个参数与 socket_read 函数相同，第三个参数指的是要读取数据的类型，默认为 PHP_BINARY_READ，也就是安全读取二进制数据；另外一个值可以是 PHP_NORMAL_READ，当读取到“\r”或“\n”换行符时停止。

8. socket_set_option

此函数用来设置 Socket 的控制选项，函数格式如下：

```
bool socket_set_option ( resource $socket , int $level , int $optname , mixed $optval )
```

例如，用该函数来设置发送超时时间为 2 s，接收超时时间为 3 s，如代码清单 5-2 所示：

代码清单5-2　使用socket_set_option函数

```
$timeout = array('sec'=>2,'usec'=>500000);
socket_set_option($socket,SOL_SOCKET,SO_RCVTIMEO,$timeout);
var_dump(socket_set_option($socket,SOL_SOCKET,SO_RCVTIMEO));
$timeout = array('sec'=>3,'usec'=>500000);
socket_set_option($socket,SOL_SOCKET,SO_SNDTIMEO,$timeout);
var_dump(socket_set_option($socket,SOL_SOCKET,SO_SNDTIMEO));
```

9. socket_last_error

该函数返回任何 Socket 操作中的函数生成的最后错误信息，其返回值为一个整型值的错误代号。完整格式如下：

```
int socket_last_error ([ resource $socket ] )
```

这个函数会帮助我们在处理错误时找到原因。下面我们一起分析 PHP 源代码的 Socket 部分的 C 源程序。

5.1.4　PHP 的 Socket 源码解析

下面我们以 socket_create 函数的源码实现来说明 PHP 的内部实现。

前面说到 PHP 的 Socket 是以扩展的方式实现的。在 PHP 源码的 ext 目录，我们找到 Sockets 目录。这个目录存放了 PHP 对于 Socket 的实现。直接搜索 PHP_FUNCTION (socket_create)，在 sockets.c 文件中找到了此函数的实现如代码清单 5-3 所示：

代码清单5-3　PHP源码中的Socket实现

```
/* {{{ proto resource socket_create(int domain, int type, int protocol) U
   Creates an endpoint for communication in the domain specified by domain, of
       type specified by type */
PHP_FUNCTION(socket_create)
{
        long            arg1, arg2, arg3;
        php_socket      *php_sock = (php_socket*)emalloc(sizeof(php_socket));

        if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "lll", &arg1,
            &arg2, &arg3) == FAILURE) {
                efree(php_sock);
                return;
        }

        if (arg1 != AF_UNIX
#if HAVE_IPV6
&& arg1 != AF_INET6
#endif
&& arg1 != AF_INET) {
                php_error_docref(NULL TSRMLS_CC, E_WARNING, "invalid socket
                    domain [%ld] specified for argument 1, assuming AF_INET",
                    arg1);
                arg1 = AF_INET;
        }
```

```
        if (arg2 > 10) {
                php_error_docref(NULL TSRMLS_CC, E_WARNING, "invalid socket
                    type [%ld] specified for argument 2, assuming SOCK_
                    STREAM", arg2);
                arg2 = SOCK_STREAM;
        }

        php_sock->bsd_socket = socket(arg1, arg2, arg3);
        php_sock->type = arg1;

        if (IS_INVALID_SOCKET(php_sock)) {
                SOCKETS_G(last_error) = errno;
                php_error_docref(NULL TSRMLS_CC, E_WARNING, "Unable to create
                    socket [%d]: %s", errno, php_strerror(errno TSRMLS_CC));
                efree(php_sock);
                RETURN_FALSE;
        }

        php_sock->error = 0;
        php_sock->blocking = 1;

1257,1-8      61%
        ZEND_REGISTER_RESOURCE(return_value, php_sock, le_socket);
}
/* }}} */
```

从代码中我们看到，Zend引擎实际是引用了系统的socket函数后进行了封装、提供给PHP调用。若你了解或学过操作系统，可以自行使用C语言对Socket调用。如代码清单5-4所示。

代码清单5-4　在C语言中初始化Socket

```
//初始化Socket
if( (socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1 ){
printf("create socket error: %s(errno: %d)\n",strerror(errno),errno);
exit(0);
}
```

从PHP内部源码来看，PHP提供的Socket编程是在socket、bind、listen等函数外添加了一个层，让其更加简单和方便调用。但是一些业务逻辑的程序还是需要开发者自己去实现。

深入Socket的内部实现机制需要我们花大量时间，感兴趣的读者可以阅读更多PHP的C实现代码。如果你不是一个底层开发者，只需要知道创建和调用已经封装好的函数

就可以了。

5.1.5　创建 TCP Socket 客户端

前面我们详细了解了 Socket 函数家族，下面就开始创建客户端（Client）和服务器（Server）。这两个术语分别代表了两种角色。

客户端是通信的发起者，而服务器程序则被动等待客户端发起通信，并对其做出响应。客户端和服务器组成了应用程序。接下来我们就创建一个 TCP Socket 客户端。

1. 连接 TCP 服务器

要连接到一个服务器上只需要两件事：IP 地址和端口号。其中 IP 地址须是真实的，也可以写解析正常的域名，这里以连接 21cto.com 为例。

我们使用 socket_connect 函数，它需要 socket 句柄和 IP 地址这两个参数去正常连接。以下为一个完整的 Socket 客户端代码示例，如代码清单 5-5 所示：

代码清单5-5　使用Socket函数族连接TCP服务器

```
if(!($socket = socket_create(AF_INET, SOCK_STREAM, 0))){
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);
    die("Couldn't create socket: [$errorcode] $errormsg \n");
}

echo "Socket is created \n";

if(!socket_connect($socket , '21cto.com' , 80)){
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Could not connect: [$errorcode] $errormsg \n");
}

echo "Connection established \n";
```

在上面的代码中使用 socket_create 函数创建一个 Socket 连接，试着使用 socket_connect 连接到一个 80 端口的 TCP 服务器。如果连接不成功，说明此端口并未打开，根据这个逻辑，我们可以完整地写出一个端口扫描的程序。

当运行此脚本时，结果类似于如下：

```
php ./socket.php
Socket created
```

```
Connection established
```

我们看到，上面的结果表示已经连接成功了。

在脚本的连接类型中，我们使用了SOCK_TREAM/TCP，这种类型的连接表示可靠的数据流，可以有多个这样的数据流，但彼此不受干扰。而其他协议如UDP、ICMP、ARP这些则是非可靠的连接。也就是说，你可以一直接收和发送下去。

我们接着做下一件事，向服务器发送数据。

2. 发送数据

我们使用函数socket_send来发送数据，如代码清单5-6所示：

代码清单5-6 向服务器发送数据

```
if(!($socket = socket_create(AF_INET, SOCK_STREAM, 0)))
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Couldn't create socket: [$errorcode] $errormsg \n");
}

echo "Socket is created \n";

if(!socket_connect($socket , '21cto.com' , 80))
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Could not connect: [$errorcode] $errormsg \n");
}

echo "Connection established \n";
$message = "GET / HTTP/1.1\r\n\r\n";

//Send the message to the server
if( ! socket_send ( $socket , $message , strlen($message) , 0))
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Could not send data: [$errorcode] $errormsg \n");
}

echo "Message send successfully \n";
```

在上面的脚本中，我们首先连接到一个服务器地址，然后发送字符串消息“GET / HTTP / 1.1 \r\n\r\n”。该消息实际上是用一个 HTTP 命令来获取一个网站的首页。我们已经发送一些数据，会收到服务器端的回复。在实际应用中需要将返回的内容写入文件中。

5.1.6　创建 TCP Socket 服务器

我们刚才是连接其他服务器，那么如何建立自己的服务器呢？如果你学过 Java 或 C，似乎它们做这样的事比较合适，PHP 适合做客户端，其实不然。接下来我们就用 PHP 来创建一个 TCP Socket 服务器，并且确保所有的数据被完整接收。

一起来看如代码清单 5-7 所示的脚本清单，注意我们在源代码加了说明文字，请你留意。

代码清单5-7　一个完整的Socket服务器

```
<?php
set_time_limit(0);
// 服务器端口与IP
$address='127.0.0.1';
$port = 6789;
// 创建一个TCP 流Socket服务器
$socket= socket_create(AF_INET, SOCK_STREAM,0);// 0 参数指为 SQL_TCP
// 绑定socket到IP或端口
socket_bind($socket,0,$port) or die('Could not bind to address');//0 适合于
localhost
// 开始侦听连接
socket_listen($socket);
if (!$socket) {
   echo "Failed to create socket!\n";
   exit;
}
//开始循环处理，等待客户端连接
while (true) {
      $client = socket_accept($socket);
      $welcome = "Welcome to the My Socket Server.Type '!exit' to close this
          connection,  or type '!die' to halt the server.";
      socket_write($client,$welcome);
      while (true) {
          $input = trim(socket_read ($client, 256));
          if ($input == '!exit') {
            break;
          }
          if ($input == '!die') {  //使用!die命令退出
```

```
            socket_close ($client);
            break 2;
        }
        $output = strtoupper($input) . "\n";
        socket_write($client,$output);
        echo $input."\n";
    }
    socket_close ($client);
}
socket_close ($socket);
?>
```

我们可以在服务器端使用命令行方式运行，可以放在crontab中，亦可放在系统的守护进程中。在Linux下可使用如下命令：

```
chomd  a+x SocketServer.php  //置为可执行t
php  -q  SocketServer.php //使用CLI模式运行
```

以上是不是很熟悉的命令？当然，我们也可以让它运行在系统守护进程：

```
nohup  php SocketServer.php  &
```

上面的命令执行完会显示背景运行后的PID。如果想杀掉此进程，可以使用如下命令：

```
Kill  pid_number
```

下面我们用telnet方式连接该端口的Socket服务器：

```
telnet 21cto.com 6789
Trying 115.28.174.91...
Connected to 21cto.com.
Escape character is '^]'.
Welcome to the My Socket Server.Type '!exit' to close this connection,  or type
    '!die' to halt the server.

Hello  //此处为手动键入的字符
HELLO
!die
Connection closed by foreigh host.
```

我们会在客户端连接到刚才创建的Socket服务器，可以看到终端中显示的欢迎信息，你输入的字会被转换为大写，使用!die退出交互和数据传输。

值得一提的是，如果你只创建一个TCP服务器实例，可以直接使用函数socket_create_listen()。

5.1.7　创建 UDP 服务器

UDP Socket 服务因为连接次数较少，处理比 TCP 简单，一个 UDP 服务器只需等待接收少量数据，与客户端在一个 Socket 中发送数据而无须连接。

下面是一个完整的 UDP 服务器的脚本代码，我们把说明文字放在了源代码中，如代码清单 5-8 所示：

代码清单5-8　UDP Socket服务器

```
<?php
error_reporting(~E_WARNING);

if(!($sock = socket_create(AF_INET, SOCK_DGRAM, 0)))
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Couldn't create socket: [$errorcode] $errormsg \n");
}

echo "Socket created \n";

// 绑定源地址与端口9999
if( !socket_bind($sock, "0.0.0.0" , 9999) )
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Could not bind socket : [$errorcode] $errormsg \n");
}

echo "Socket bind OK \n";

//开始处理通信，循环处理多客户端
while(1)
{
    echo "Waiting for data ... \n";

    //接收一些数据
    $r = socket_recvfrom($sock, $buf, 512, 0, $remote_ip, $remote_port);
    echo "$remote_ip : $remote_port -- " . $buf;

    //返回数据给客户端
    socket_sendto($sock, "OK " . $buf , 100 , 0 , $remote_ip , $remote_port);
```

```
}

socket_close($sock);
```

可以看到，以上脚本会在本地 localhost 主机的 9999 端口创建一个 UDP 服务，我们在服务器端运行这个程序，如图 5-4 所示。

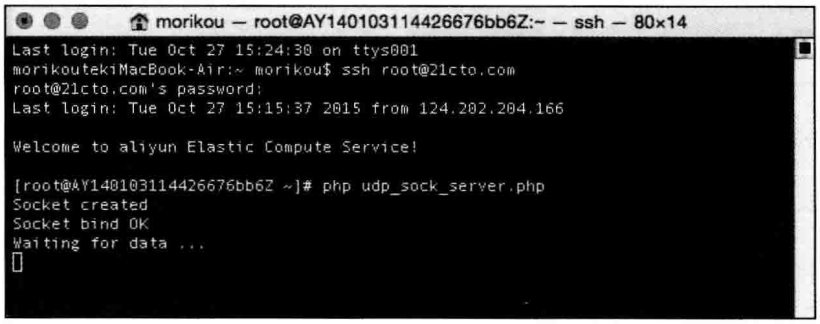


图 5-4　建立本地 UDP 服务器

这个 UDP 服务器可以处理多个客户，但只是简单地处理输入后的返回信息，感兴趣的话你可以在此基础上继续扩展功能，如精确处理指令等。

创建 UDP Socket 客户端

从上面内容来看，我们的 UDP 服务器运行良好，它能处理输入并返回信息，有这样的功能足够我们测试。接下来写一个客户端程序，让它与服务器之间进行数据通信，如代码清单 5-9 所示：

代码清单5-9　UDP Socket客户端

```
<?php
error_reporting(~E_WARNING);
$server = '127.0.0.1';
$port = 9999; //连接服务器的端口号

if(!($sock = socket_create(AF_INET, SOCK_DGRAM, 0)))
{
    $errorcode = socket_last_error();
    $errormsg = socket_strerror($errorcode);

    die("Couldn't create socket: [$errorcode] $errormsg \n");
}
```

```
echo "Socket created \n";

//循环处理通信
while(1)
{
    //显示数据输入界面
    echo 'Enter a message to send : ';
    $input = fgets(STDIN);

    //发送信息给服务器端
    if( ! socket_sendto($sock, $input , strlen($input) , 0 , $server , $port))
    {
        $errorcode = socket_last_error();
        $errormsg = socket_strerror($errorcode);

        die("Could not send data: [$errorcode] $errormsg \n");
    }

    //现在从服务器接收回复并打印
    if(socket_recv ( $sock , $reply , 2045 , MSG_WAITALL ) === FALSE)
    {
        $errorcode = socket_last_error();
        $errormsg = socket_strerror($errorcode);

        die("Could not receive data: [$errorcode] $errormsg \n");
    }

    echo "Reply : $reply";
}
```

代码执行结果如图 5-5 所示。

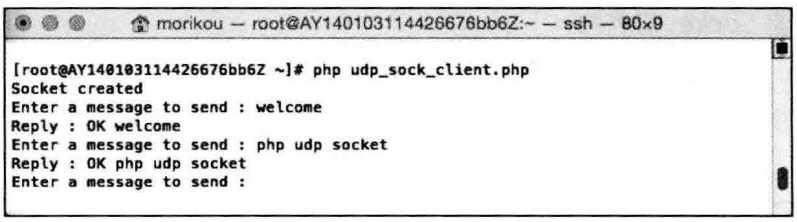


图 5-5　代码执行结果

与此同时也能看到服务器端接收的内容显示，如图 5-6 所示。

以上表示我们的 UDP 服务器与客户端已经建立连接，并且能够交互成功了。

图 5-6 服务器端接收的内容

5.1.8 字符流与 Socket

1. 字符流 Socket 客户端

另外 PHP 还提供了字符流方式处理（Stream），比如把文件、字符串转换为二进制流。接下来让我们看一下使用流方式处理 Socket 的方法。

有一个函数叫 stream_socket_clien，它的功能类似于 fsockopen，如代码清单 5-10 所示：

代码清单5-10 使用流方式连接Socket服务器

```
$fp = stream_socket_client("tcp://www.google.com:80", $errno, $errstr, 30);
if (!$fp) {
    echo "$errstr ($errno)<br />n";
}
```

我们可以看到这个 Socket 网址和前面的服务器地址很像，但唯一区别的是需要增加一个端口号。

2. 字符流 Socket 服务器

上面我们用流方式编写了 Socket 客户端。下面我们再来写 Socket 服务器端，接收 Socket 连接并回复请求数据，如代码清单 5-11 所示。

代码清单5-11 字符流方式Socket服务器

```
$server = stream_socket_server("tcp://0.0.0.0:4444", $errno, $errorMessage);
if ($server === false)
{
    throw new UnexpectedValueException("Could not bind to socket: $errorMessage");
}
```

```
for (;;)
{
    $client = stream_socket_accept($server);

    if ($client)
    {
        echo 'Connection accepted from '.stream_socket_get_name($client, false) .
            "n";

        stream_copy_to_stream($client, $client);
        fclose($client);
    }
}
```

我们在系统终端运行以上 PHP 脚本，即开始运行 Socket 服务器，代码如下：

```
$telnet localhost 4444
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
sd
sd
```

这个服务器实现了一个简单的 echo 服务器，实现了发送与回复功能。但这个服务器有一个局限，不能处理多个连接。如果有一个客户端已经连接，就不能再处理其他的连接了。

下面我们来看一下如何实现处理多个客户端的代码，如代码清单 5-12 所示：

代码清单5-12　处理多客户端的Socket服务器

```
<?php
// 开放一个服务器端口 4444
$server = stream_socket_server("tcp://0.0.0.0:4444", $errno, $errorMessage);

if ($server === false)
{
    die("Could not bind to socket: $errorMessage");
}

$client_socks = array();
while(true)
{
    //准备读取Socket数据
    $read_socks = $client_socks;
    $read_socks[] = $server;
```

```
    //开始读取，使用非常长的超时时间
    if(!stream_select ( $read_socks, $write, $except, 300000 ))
    {
        die('something went wrong while selecting');
    }

    //新客户端
    if(in_array($server, $read_socks))
    {
        $new_client = stream_socket_accept($server);

        if ($new_client)
        {
            //打印远程客户端信息，包括IP和端口
            echo 'Connection accepted from ' . stream_socket_get_name($new_
                client, true) . "n";

            $client_socks[] = $new_client;
            echo "Now there are total ". count($client_socks) . " clients.n";
        }

        //从已读sockets中删除该服务器sockets
        unset($read_socks[ array_search($server, $read_socks) ]);
    }

    //从已存在的客户端取信息
    foreach($read_socks as $sock)
    {
        $data = fread($sock, 128);
        if(!$data)
        {
            unset($client_socks[ array_search($sock, $client_socks) ]);
            @fclose($sock);
            echo "A client disconnected. Now there are total ". count($client_
                socks) . " clients.n";
            continue;
        }
        //发送返回信息给客户端
        fwrite($sock, $data);
    }
}
```

该 stream_select() 函数处理 Socket 的读写或错误，放在一个数组中。格式如下：

```
stream_select ( $read_socks, $write, $except, 300000 )
```

当 Socket 发生一个特定事件时，stream_select 将 Socket 返回的信息放在数组中。例如，如果 Socket 接收到客户端消息，stream_select 将内容放在 read_socks 数组并返回。同样，有新的连接也将导致 stream_select 返回。这种情况下，在 Socket 数组中没有任何内容。

我们使用如下命令行操作：

```
$ php /var/www/socket.php
Connection accepted from 127.0.0.1:42203
Now there are total 1 clients.
Connection accepted from 127.0.0.1:42204
Now there are total 2 clients.
Connection accepted from 127.0.0.1:42205
Now there are total 3 clients.
A client disconnected. Now there are total 2 clients.
A client disconnected. Now there are total 1 clients.
A client disconnected. Now there are total 0 clients.
```

服务器控制台上的输出会显示有 3 个客户端连接和断开的状态。

5.1.9　连接 SMTP 服务器

在开发 Web 应用时，都会用到邮件的发送或接收。PHP 自带有一个 mail 函数，如果运行环境安装了 Postfix 或 Qmail 之类的 MTA（Mail Transfer Agent），使用这个函数当然没有问题，如果没有安装，使用这个函数则不能成功发送邮件。

因此，需要借助外部 SMTP 邮件服务器来发送邮件，php.ini 中配置 SMTP 的选项，而我从来没有这样配置过。如果服务器有多个应用，发件人将可能是一个地址，这个选项形同虚设。

另外，也可以使用 PEAR、phpMailer、SwiftMailer 等第三方类库来发邮件。这些类库也提供了 SMTP 的方法来发送邮件。

在这一节里，我们可以不用这些略显复杂的类，即使用刚刚掌握的 Socket 功能连接这些邮件提供商，如网易、腾讯、Gmail 的 SMTP 服务器来发送邮件。

1. 什么是 SMTP 协议

我们先来重温一下 SMTP 原理。SMTP（Simple Mail Transfer Protocol）是简单邮件传输协议，它是一组用于由源地址到目的地地址传送邮件的规则，由它来控制信件的中转方式。

SMTP 协议属于 TCP/IP 协议簇，它帮助每台计算机在发送或中转信件时找到下一个目的地。通过 SMTP 协议所指定的服务器，就可以把 e-mail 寄到收信人的服务器上了，

整个过程只要几分钟。SMTP 服务器则是遵循 SMTP 协议的发送邮件服务器，用来发送或中转发出的电子邮件。它使用由 TCP 提供的可靠的数据传输服务把邮件消息从发信人的邮件服务器传送到收信人的邮件服务器。跟大多数应用层协议一样，SMTP 也存在两个端：在发信人的邮件服务器上执行的客户端和在收信人的邮件服务器上执行的服务器端。

SMTP 的客户端和服务器端同时运行在每个邮件服务器上。当一个邮件服务器在向其他邮件服务器发送邮件消息时，它是作为 SMTP 客户在运行的。

SMTP 协议与人们用于面对面交互的礼仪之间有许多相似之处。运行步骤如下：

首先，运行在发送端邮件服务器主机上的 SMTP 客户，发起建立一个到运行在接收端邮件服务器主机上的 SMTP 服务器端口号 25 之间的 TCP 连接。

如果接收邮件服务器当前不在工作，SMTP 客户就等待一段时间后再尝试建立该连接。SMTP 客户和服务器先执行一些应用层握手操作，就像人们在转手东西之前往往先自我介绍那样，SMTP 客户和服务器也在传送信息之前先自我介绍一下。

在这个 SMTP 握手阶段，SMTP 客户向服务器分别指出发信人和收信人的电子邮件地址。彼此自我介绍完毕之后，客户发出邮件消息。

2. 命令行实践

了解了以上 SMTP 原理，下面我们就以实践的方式使用 SMTP 命令来发送一封邮件。

第一步，远程登录 SMTP 服务器。我们在命令行窗口输入：

```
telnet smtp.126.com 25
```

此处以 126（163 亦如同例）邮箱为例，QQ 邮箱可输入 smtp.qq.com，25 表示建立连接的端口号。然后回车，会出现如图 5-7 所示内容。

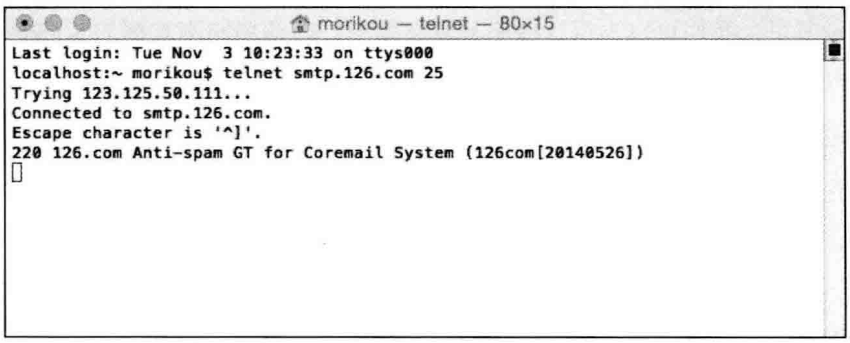


图 5-7 远程登录 SMTP 服务器

如上的信息为一个欢迎信息，各个邮箱运营商的提示有所不同。

第二步，进入用户登录。输入 helo 126.com 并回车，这是向服务器表明你的用户身份，如图 5-8 所示。

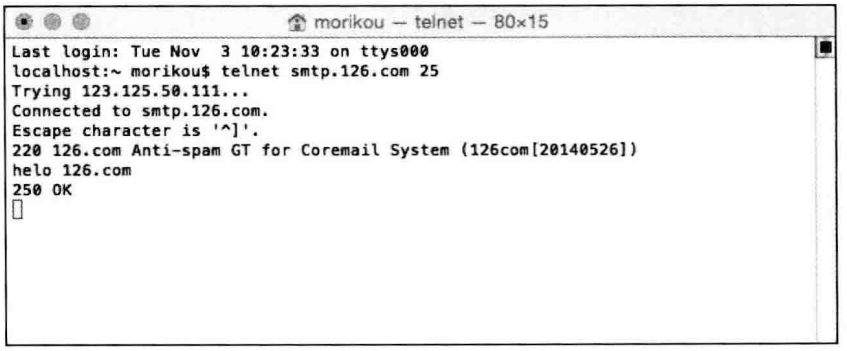


图 5-8　用户登录

需要注意的是，此时的字母是不能输错的，因为每一次按键已经通过 Socket 传送到服务器，所以输入错误时不能使用退格键删除，只能换行重新输入。

250 OK 为服务器返回的确认信息，说明服务器认可了你的身份。

输入 auth login 并回车，这是告诉服务器你要输入用户名了（即登录邮箱时的用户名），服务器将返回一个确认信息 334，如图 5-9 所示。

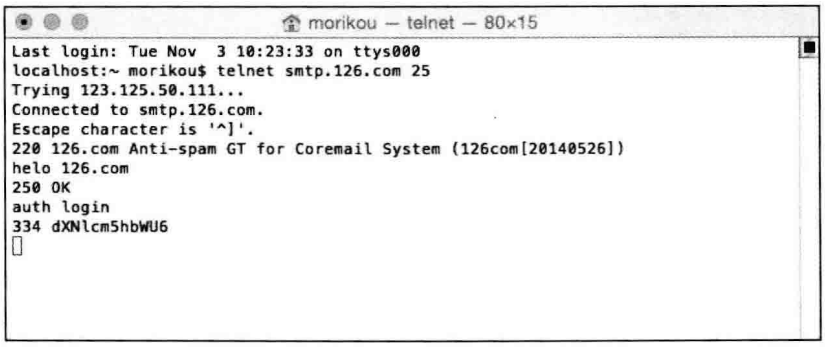


图 5-9　服务器返回确认信息 334

这时可以输入用户名了，注意用户名需要编码后才能识别，需要将用户名 dujiang 进行 base64 编码，将编码结果输入命令提示符窗口。我们可以使用 base64_decode() 函数来处理。代码如下：

```
echo base64_decode('dujiang');
```

输入编码后的用户名，回车后提交 SMTP 服务器鉴权。会出现如图 5-10 所示之提示。

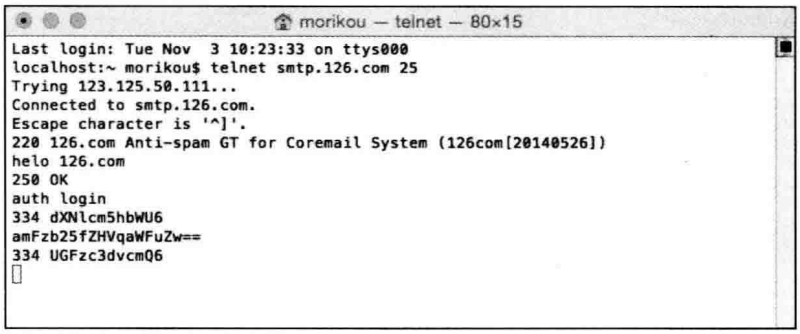


图 5-10 输入用户名后提交 SMTP 服务器鉴权

用户名输入成功后，服务器返回一个 334 的确认信息。输入成功后，接着输入密码，密码同样要经过 base64 编码后再输入，如图 5-11 所示。

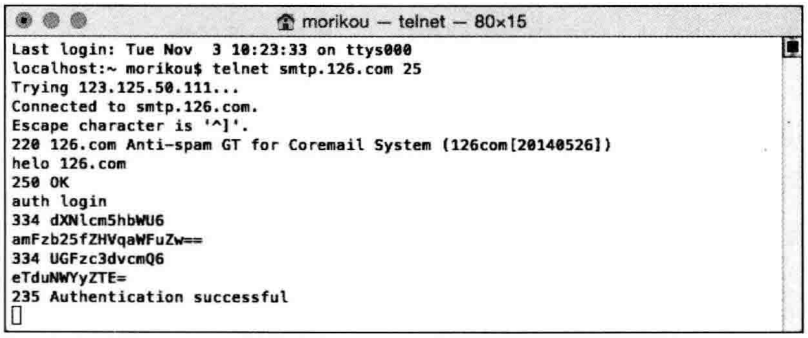


图 5-11 输入密码

输入成功后服务器返回一个含 successfully 的信息，说明成功登录。

第三步，开始写信。

输入 mail from：回车，在 <> 中填写发件人的邮箱，即你自己的邮箱名。

输入 rcpt to：回车，在 <> 中填入收件人的邮箱名。

输入 data：回车，回车后就可以开始写邮件内容了。

下面是 data 下的可选项：

输入 from：发件人名称，此项可任意填入，将显示在收件箱的“发件人”一栏。

输入 to：收件人名称，可任意填入，将显示在收件箱的“收件人”一栏；

输入 subject：信件主题，将显示在收件箱的“主题”一栏。

此时需空一行，即在一空行直接回车，表示正文部分的开始。空行后输入信件的正文内容。

在正文输入结束时输入一个 .（英文输入法下的句号），回车，表示正文部分的结束。这时将显示邮件成功发送的信息。图 5-12 所示是发信的完整图示，作为参考。

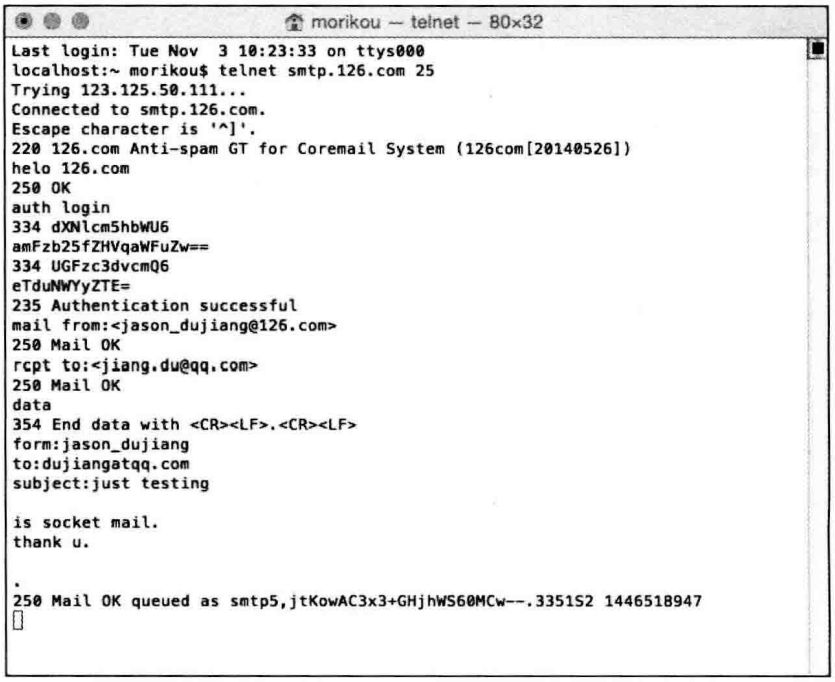


图 5-12　发信的完整图示

正如你所看到的，SMTP 服务器提示已经加入队列，此时就可以检查收件人的邮件，如果信找不到，可以在垃圾邮件里找一下。图 5-13 表示发送邮件已经成功。本例中，我使用了腾讯 QQ 的邮箱。如图 5-13 所示。

图 5-13 为通过命令行收到的邮件，也可使用其他邮件提供商尝试。下面我们自己编写一个 SMTP 发送邮件的脚本，用于发送不同格式的邮件。

3. 发送邮件

上面我们已经通过命令行可以发邮件了，那么我们开发一个发邮件的程序也就不难了，当然还有一些处理错误，增加选项，多种邮件格式用命令行发送的话就太辛苦了。所以我们需要开发一个 SMTP 邮件类，以后只需要传递一些参数就可以方便地发送邮件。下面是发送邮件的完整清单，一些解释说明也请留意，如代码清单 5-13 所示：

图 5-13 QQ 邮箱通过命令收到的邮件

代码清单5-13 SMTP发送类

```
<?php
//SMTP邮件发送类
class smtp
{
    public $smtp_port;
    public $time_out;
    public $host_name;
    public $log_file;
    public $relay_host;
    public $debug;
    public $auth;
    public $user;
    public $pass;
    /* 私有变量*/
    public $sock;

    /* 构造方法 */
    function smtp($relay_host = "", $smtp_port = 25,$auth = false,$user,$pass)
```

```
{
        $this->debug = FALSE;
        $this->smtp_port = $smtp_port;
        $this->relay_host = $relay_host;
        $this->time_out = 30; //is used in fsockopen()
        $this->auth = $auth;
        $this->user = $user;
        $this->pass = $pass;
        $this->host_name = "localhost"; //is used in HELO command
        $this->log_file = "";
        $this->sock = FALSE;
}

/* 邮件发送*/
function sendmail($to, $from, $subject = "", $body = "", $mailtype, $cc = "",
    $bcc = "", $additional_headers = "")
{
        $mail_from = $this->get_address($this->strip_comment($from));
        $body = ereg_replace("(^|(\r\n))(\.)", "\1.\3", $body);
        $header = "MIME-Version:1.0\r\n";
        if($mailtype == "HTML"){
                $header .= "Content-Type:text/html\r\n";
        }
        //收件人地址
        $header .= "To: ".$to."\r\n";
        //抄送地址
        if ($cc != "") {
                $header .= "Cc: ".$cc."\r\n";
        }
        //构造邮件头
        $header .= "From: $from<".$from.">\r\n";
        $header .= "Subject: ".$subject."\r\n";
        $header .= $additional_headers;
        $header .= "Date: ".date("r")."\r\n";
        $header .= "X-Mailer:By PHPMailer (PHP/".phpversion().")\r\n";
        list($msec, $sec) = explode(" ", microtime());
        $header .= "Message-ID: <".date("YmdHis", $sec).".".($msec*1000000).
            ".".$mail_from.">\r\n";

        $TO = explode(",", $this->strip_comment($to));

        if ($cc != "") {

                $TO = array_merge($TO, explode(",", $this->strip_comment($cc)));
```

```
    }
    if ($bcc != "") {
        $TO = array_merge($TO, explode(",", $this->strip_comment($bcc)));
    }
    $sent = TRUE;
    foreach ($TO as $rcpt_to) {
        $rcpt_to = $this->get_address($rcpt_to);
        if (!$this->smtp_sockopen($rcpt_to)) {
            $this->log_write("Error: Cannot send email to ".$rcpt_
                to."\n");
            $sent = FALSE;
            continue;
        }
        if ($this->smtp_send($this->host_name, $mail_from, $rcpt_to,
            $header, $body)) {
            $this->log_write("E-mail has been sent to <".$rcpt_
                to.">\n");
        } else {
            $this->log_write("Error: Cannot send email to
                <".$rcpt_to.">\n");
            $sent = FALSE;
        }
        fclose($this->sock);
        $this->log_write("Disconnected from remote host\n");
    }
```

```
        return $sent;

}

/* 私有方法 */
function smtp_send($helo, $from, $to, $header, $body = "")
{
        if (!$this->smtp_putcmd("HELO", $helo)) {
                return $this->smtp_error("sending HELO command");
        }
        //认证过程
        if($this->auth){
                if (!$this->smtp_putcmd("AUTH LOGIN", base64_encode($this->
                    user))) {
                        return $this->smtp_error("sending HELO command");
                }
                if (!$this->smtp_putcmd("", base64_encode($this->pass))) {
                        return $this->smtp_error("sending HELO command");
                }
        }

        if (!$this->smtp_putcmd("MAIL", "FROM:<".$from.">")) {
                return $this->smtp_error("sending MAIL FROM command");
        }

        if (!$this->smtp_putcmd("RCPT", "TO:<".$to.">")) {
                return $this->smtp_error("sending RCPT TO command");
        }

        if (!$this->smtp_putcmd("DATA")) {
                return $this->smtp_error("sending DATA command");
        }

        if (!$this->smtp_message($header, $body)) {
                return $this->smtp_error("sending message");
        }

        if (!$this->smtp_eom()) {
                return $this->smtp_error("sending <CR><LF>.<CR><LF> [EOM]");
        }

        if (!$this->smtp_putcmd("QUIT")) {
                return $this->smtp_error("sending QUIT command");
        }
```

```
        return TRUE;
}

function smtp_sockopen($address)
{
        if ($this->relay_host == "") {
                return $this->smtp_sockopen_mx($address);
        } else {
                return $this->smtp_sockopen_relay();
        }
}

function smtp_sockopen_relay()
{
        $this->log_write("Trying to ".$this->relay_host.":".$this->smtp_
            port."\n");
        $this->sock = @fsockopen($this->relay_host, $this->smtp_port,
            $errno, $errstr, $this->time_out);
        if (!($this->sock && $this->smtp_ok())) {
                $this->log_write("Error: Cannot connenct to relay host ".$this->
                    relay_host."\n");
                $this->log_write("Error: ".$errstr." (".$errno.")\n");
                return false;
        }
        $this->log_write("Connected to relay host ".$this->relay_host."\n");
        return true;
}
//检查域名MX记录是否正确
function smtp_sockopen_mx($address)
{
        $domain = ereg_replace("^.+@([^@]+)$", "\1", $address);
        if (!@getmxrr($domain, $MXHOSTS)) {
                $this->log_write("Error: Cannot resolve MX \"".$domain."\"\n");
                return FALSE;
        }

        foreach ($MXHOSTS as $host) {
                $this->log_write("Trying to ".$host.":".$this->smtp_port."\n");
                $this->sock = @fsockopen($host, $this->smtp_port, $errno,
                    $errstr, $this->time_out);
                if (!($this->sock && $this->smtp_ok())) {
                        $this->log_write("Warning: Cannot connect to mx host
                            ".$host."\n");
                        $this->log_write("Error: ".$errstr." (".$errno.")\n");
                        continue;
```

```
                }
                $this->log_write("Connected to mx host ".$host."\n");
                return TRUE;
        }
        $this->log_write("Error: Cannot connect to any mx hosts (".implode
            (", ", $MXHOSTS).")\n");
        return FALSE;
}

function smtp_message($header, $body)
{
        fputs($this->sock, $header."\r\n".$body);
        $this->smtp_debug("> ".str_replace("\r\n", "\n."> ", $header."\n>
            ".$body."\n> "));
        return TRUE;
}

function smtp_eom()
{
        fputs($this->sock, "\r\n.\r\n");
        $this->smtp_debug(". [EOM]\n");
        return $this->smtp_ok();
}

function smtp_ok()
{
        $response = str_replace("\r\n", "", fgets($this->sock, 512));
        $this->smtp_debug($response."\n");
        if (!ereg("^[23]", $response)) {
                fputs($this->sock, "QUIT\r\n");
                fgets($this->sock, 512);
                $this->log_write("Error: Remote host returned \"".$response."\
                    "\n");
                return FALSE;
        }
        return TRUE;
}

function smtp_putcmd($cmd, $arg = "")
{
        if ($arg != "") {
                if($cmd=="") $cmd = $arg;
        else $cmd = $cmd." ".$arg;
        }
        fputs($this->sock, $cmd."\r\n");
```

```
        $this->smtp_debug("> ".$cmd."\n");
        return $this->smtp_ok();
}

function smtp_error($string)
{
        $this->log_write("Error: Error occurred while ".$string.".\n");
        return FALSE;
}
//记录日志
function log_write($message)
{
        $this->smtp_debug($message);
        if ($this->log_file == "") {
                return TRUE;
        }

        $message = date("M d H:i:s ").get_current_user()."[".getmypid()."]:
            ".$message;

        if (!@file_exists($this->log_file) || !($fp = @fopen($this->log_file,
            "a"))) {
                $this->smtp_debug("Warning: Cannot open log file \"".$this->
                    log_file."\"\n");
                return FALSE;;
        }
        flock($fp, LOCK_EX);
        fputs($fp, $message);
        fclose($fp);
        return TRUE;
}

function strip_comment($address)
{
        $comment = "\([^()]*\)";
        while (ereg($comment, $address)) {
                $address = ereg_replace($comment, "", $address);
        }

        return $address;
}

function get_address($address)
{
        $address = ereg_replace("([ \t\r\n])+", "", $address);
```

```
            $address = ereg_replace("^.*<(.+)>.*$", "\1", $address);
            return $address;
    }

    function smtp_debug($message)
    {
            if ($this->debug) {
                    echo $message;
            }
    }
}
```

以上的SMTP类是一个完整的邮件发送库，在处理Socket过程中均做了日志处理，以方便调试，可以发送纯文本和混合型（mixed）类邮件，这属于邮件MIME定义范围之内，并且可以判断邮件地址是否有效，等等，在这里不做赘述，请着重关注Socket处理的部分即可。

以下是调用上面SMTP类的方法，如代码清单5-14所示：

代码清单5-14 使用SMTP协议发送邮件

```
<?php
    require_once "email.class.php";
    $smtpserver = "smtp.126.com";//SMTP服务器
    $smtpserverport =25;//SMTP服务器端口
    $smtpusermail = "jacky";//SMTP服务器的用户邮箱
    $smtpemailto = "194160099@qq.com";//收件人
    $smtpuser = "just";//SMTP服务器的用户账号
    $smtppass = "youpass";//SMTP服务器的用户密码
    $mailtitle = "hello world";//邮件主题
    $mailcontent = "<h1>this is test mailer</h1>";//邮件内容
    $mailtype = "HTML";//邮件格式（HTML/TXT）,TXT为文本邮件

    $smtp = new smtp($smtpserver,$smtpserverport,true,$smtpuser,$smtppass);//这
        里面的一个true是表示使用身份验证,否则不使用身份验证
    $smtp->debug = false;//是否显示发送的调试信息

    $state = $smtp->sendmail($smtpemailto, $smtpusermail, $mailtitle, $mail-
        content, $mailtype);

    if($state==""){
            echo "对不起，邮件发送失败！请检查邮箱填写是否有误。";
            exit();
    }
    echo "恭喜！邮件发送成功！！";
```

和前面一样的测试，我们可查收到该邮件表示发送成功。

可以看到，依靠纯 PHP 无法完成的任务，使用 Socket 功能可以完全实现。一些软件系统的底层也可以依照此原理，由其提供 Socket 服务接口，用 PHP 请求该服务，完成产品的功能，如支付接口、游戏数据接口、证券财务服务等。

5.2 cURL 核心技术

使用 cURL 可以实现网络数据的抓取，是一个包含诸多选项的强大接口。有了这些功能我们能够轻易地开发搜索引擎的爬虫（spiders 或 webbots）或自动化传输程序，如价格监控、图片抓取、链接校验、搜索排名检测、信息聚合等。

编写爬虫类的程序有趣且充满挑战，有趣的是发现你可能从未发现的新事情，有挑战的是就是对方防止抓取时或者技术上的安全考量。

5.2.1 什么是 cURL

cURL 是 client URL Library Functions 的缩写。它的底层是由一个命令行工具实现的，用于获取远程文件或传输文件，支持 FTP/FTPS、HTTP/HTTPS、SCP/SFTP、Telnet、DICT 和 File/LDAP 等协议，通俗来讲，些类型的服务器，cURL 都可以抓取。

cURL 的本质是一个命令行的工具，如模拟一个正常用户在浏览器上访问一个网站，可以实现自动登录，或者自动抓取分析网页中的内容、二进制文件数据等。

cURL 是由瑞典的开发者 Daniel Stenberg 和一个开发小组用 C 开发的，名字称为 LibCURL，是一个开源软件，遵守 MIT 协议。cURL 相当于一个库 /API，可单独打包到自己的应用程序中，所以并不是只有 PHP 语言才能使用 cURL。

有经验的程序员会问，我们为何没有用 file_get_contents() 这个函数来读取远端内容？实际上这个函数是 file() 函数的一系列包装，而且性能方面存在比较大的问题，在执行时会非常慢。此外我们也没有办法在读取过程中有效地进行监控和错误处理，还有 file_get_contents() 函数没有办法完成更复杂些的任务，比如管理 Cookies，验证和表单提交，文件上传，等等，这也是我们使用 cURL 的重要原因。

5.2.2 安装和启用 cURL

如果你用的是 Windows 系统，在 php.ini 文件中找到以下内容，去掉分号，然后重启你的 Nginx 或 Apache 服务器。

```
;extension=php_curl.dll
```

如果你用的是 Ubuntu 系统，使用 apt-ge 命令安装 curl，之后请重启 Web 服务器。

```
sudo apt-get install php5-curl
```

在 CentOS 下，使用 yum 命令来安装 curl，成功重启 Web 服务器后也即时启用 cURL。

```
yum install php-curl
```

我们可以用以下代码检查是否可以用 cURL 扩展库。

```
if(is_callable('curl_init')){
    echo"curl is enabled";
}
else
{
    echo"curl not enabled";
}
```

也可使用 phpinfo() 函数来查看，这个画面会让我们了解更丰富一点儿的内容，比如版本号、支持的选项等，如图 5-14 所示。

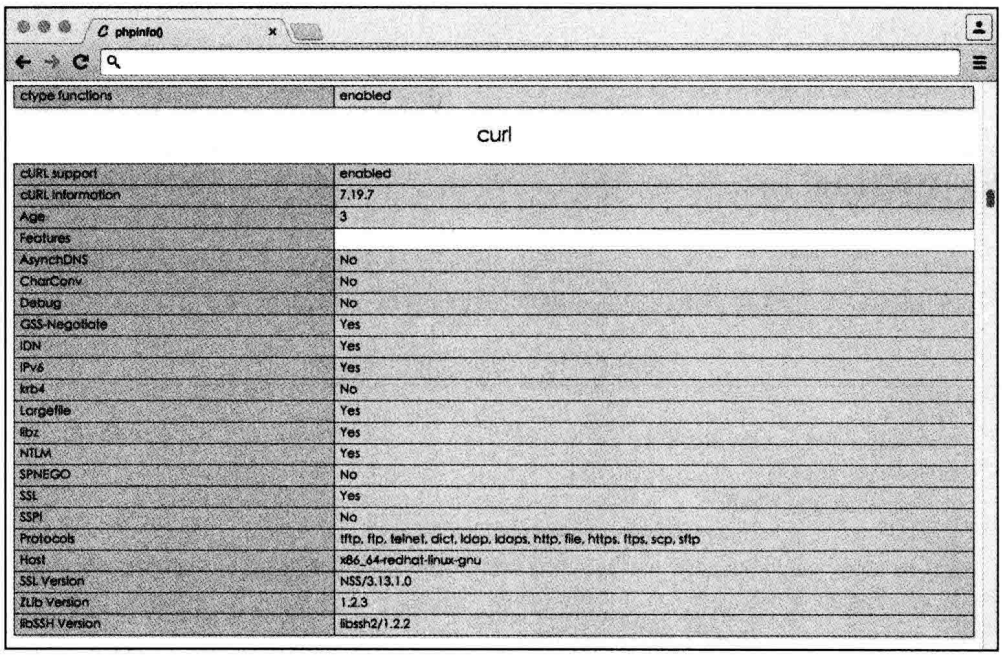


图 5-14　cCURL 库安装成功界面

如果在 PHP 安装编译时添加了 cURL，那么可以忽略上面的介绍。

5.2.3 建立 cURL 的步骤

cURL 和文件处理非常类似，两者都创建了指向外部文件的句柄，两者在传送完成后都会自动关闭。cURL 与文件 I/O 不同的是，需要定义文件传输的属性选项。下面来了解 cURL 执行的步骤。

首先我们使用 curl_init 启动一个 cURL 实例：

```
$curl = curl_init( “www.21cto.com” );  //定义目标网站
```

这个 curl_init 函数将返回一个句柄给 $curl 变量。接下来，我们会设置该句柄的选项，包括 URL，语法如下：

```
curl_setopt($curl,CONSTANT,Value);
```

这个选项告诉 cURL 如何初始化会话、设置选项，这个参数多得令人咂舌，超过 90 多项，因此该接口的功能非常强大。不过不用为此有任何心理负担，我们在日常应用中使用的只是其中一小部分，这些参数除了一部分初始化外，其余与顺序无关。

当我们设置好选项后，使用 curl_exec() 函数来执行一个整个 cURL 事务，如抓取操作。

```
$result = curl_exec($curl);
```

这表示将执行后的结果放在 $result 变量中，方便我们显示和处理结果。最后，我们需释放该连接句柄。

```
curl_close($curl);
```

我们来看一个完整的 cURL 实例。脚本如代码清单 5-15 所示：

代码清单5-15 cURL完整执行实例

```
<?php
//1.初始化，创建一个新cURL句柄
$ch = curl_init();
//2.设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.21cto.com/");
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);   //设置curl_exec()获取的信息以字符流
    的形式返回
curl_setopt($ch,CURLOPT_REFERER,"http://weibo.com/");
curl_setopt($ch, CURLOPT_HEADER, 1); //启用时会将头的信息作为字符流输出
//3.抓取URL的HTML内容
```

```
$output = curl_exec($ch);
//4.关闭cURL句柄，并且释放系统资源
curl_close($ch);
?>
```

在上面的实例中，我们使用了 CURLOPT_RETURNTRANSFER 选项。当设置为 TRUE 时，PHP cURL 会将结果回显到终端上。

有时候我们并不需要 header 头内容返回，可以把 CURLOPT_HEADER 设置为 0，或者不设置，默认值就是 0。

另外该例中还使用了 CURLOPT_REFERER 选项。此选项允许开发者欺骗超链接的点击状态来初始化目标文件的请求。上面的例子是告诉目标服务器是某人通过点击了 http://weibo.com/index.php 中的链接后到达的目标网页。

当完成 cURL 会话后应该关闭。在正常情况下 PHP 会执行垃圾回收的工作，在脚本执行完毕后会释放诸如变量、Socket 连接及本地内存等资源。这就相当于页面执行后迅速结束并不会产生任何问题，但是爬虫类程序大多数需要执行一段时间并不执行垃圾回收工作。如果一个爬虫脚本中有多个 cURL 操作，则必须关闭每个 cURL 会话。

另外前面提到过，cURL 第二步的 curl_setopt 参数比较多，很多的复杂处理也都在此处，我们有必要了解一些重要的选项。

5.2.4　PHP cURL 选项

上面我们根据一个实例介绍了 CURLOPT_RETURNTRANSFER、CURLOPT_HEADER 和 CURLOPT_REFERER 3 个选项。接下来我们先把一些主要参数介绍一下，这些参数已经被遴选，一是会经常用，二是可能会出现模糊的情况。

1. CRUL_OPT_FOLLOWLOCATION 和 CURLOPT_MAXREDIRS

CURL_OPT_FOLLOWLOCATION 参数用来跟踪目标页面是否有重定向。如下例：

```
<?php
header( 'Location:http://www.21cto.com' );
?>
<meta http-equiv=" Refresh"  Content=' 0;http://www.21cto.com' >
<script type=' javascript' >document.location=' http://www.21cto.com' ;</script>
```

CURL_OPT_FOLLOWLOCATION 目前只支持 header 头重定向的处理，对于后面页面中的 mata 标识符和 js 跳转则无法识别。

CURL_OPT_MAXREDIRS 用于定义最大的重定向跟踪次数，以防止抓取过程总是

进入同一个网址，进入死循环。在此处我有过这样的教训，当多次抓取某个网站的内容时，这种情况会被对方网管注意到，他会有被人为攻击或抓取数据的判断力，来自重复的请求会拒绝访问或干脆重定向到一个 404 的页面上。如果拒绝访问，我们可以通过更换 IP 地址或通过代理服务器发起请求。如果是重定向页面，我们的爬虫程序会被引到一个歧途，导致死循环和大量的带宽使用，且数据亦可能无法获取到。

使用 CURL_OPT_MAXREDIRS 可以有效解决此问题。代码片断如下：

```
//etc
curl_setopt($ch,CURLOPT_FOLLOWCATION,TRUE); //跟随header头重定向
curl_setopt($ch,CURLOPT_MAXREDIRS,5); //设置重定向跟随为5次
```

2. CURLOPT_USERAGENT

该选项很重要，也很有用。它是用来定义用户代理的名称，比如我们模拟微信的内置浏览器访问某个网站。代码片段如下：

```
$agent = 'Mozilla/5.0 (iPhone; CPU iPhone OS 5_1 like Mac OS X) AppleWebKit/
    534.46 (KHTML, like Gecko) Mobile/9B176 MicroMessenger/4.2.2';
curl_setopt($ch,CURLOPT_USERAGENT,$agnet);
```

现在有一些网站会检测用户代理，然后再决定是否提供服务，或根据代理的不同，显示不同的页面，比如用 PC 浏览器访问某些手机或微信端网站，它会拒绝显示或者重定向到 PC 端网站。

用户代理名称随着移动端的浏览器版本升级而发生一些变化，我们需要自己取得这些标识的正确名称，比如用你的 iPhone、Android、iPad 等设备访问自己的一个服务器页面，用如下代码即可获得。

```
echo $_SERVER['HTTP_USER_AGENT'];
```

3. CURLOPT_NOBODY 与 CURLOPT_HEADER

这两个参数有一个前面我们已介绍过，前者用来设置是否返回内容的主体内容，后者用来设置是否返回页面的头内容。

以下代码用来设置为去除页面主体，返回网页的头内容：

```
curl_setopt($ch,CURLOPT_HEADER,TRUE); //返回的字符流包含标准头
curl_setopt($ch,CURLOPT_NOBODY,TRUE); //排除页面主体
```

4. CURLOPT_TIMEOUT 与 CURLOPT_CONNECTTIMEOUT

顾名思义，这个选项是用来设置 cURL 操作时等待目标服务器的响应时间，如果目

标服务器宕机，极端一些，比如管理员发现网站被抓取，一气之下关了主机电源，这样会造成我们的爬虫无限的等待。

其实此种场景经常出现在抓取一些访问量大的站点，返回较慢、链接404等。用以下代码来设置在cURL抓取操作超过规定秒数时终止会话。

```
curl_setopt($ch,CURLOPT_TIMEOUT,30); //设置等待时间为30秒以内
```

这样，即便管理员关机，我们也不会傻傻地等待。

5. CURLOPT_COOKIEFILE与CURLOPT_COOKIEJAR

这两个选项是PHP cURL库提供的强大且灵活的特性之一，它能够帮我们在传输抓取过程中向服务器传递Cookie信息。

我们可以使用CURLOPT_COOKIEFILE选项定义之前存储Cookie的文件位置。在会话完成时，cURL会把新的Cookie写入CURLOPT_COOKIEJAR指定的文件中。代码片段如下：

```
curl_setopt($ch,CURLOPT_COOKIEFILE,' /usr/tmp/cookies.txt' ); //读取cookie文件
curl_setopt($ch,CURLOPT_COOKIEJAR,' /usr/tmp/cookies.txt' ); //写入cookie文件
```

需要注意的有两点，一是文件一定要可读写，二是指定的Cookie文件的路径一定要写成绝对路径。

6. CURLOPT_HTTPHEADER

这个选项亦为比较重要的内容。它可让我们自定义在让目标服务器接收的header头内容，如告之目标服务器能接收的MIME、内容类型、用户代理及压缩类型等内容。

请看如下代码：

```
$header_arr [] = 'Mime-version:1.0' ;
$header_arr[] = 'Content-type:text/html;charset=utf-8' ;
$header_arr[] = 'Accept-encoding:compress,gzip' ;
curl_setopt($curl_session,CURLOPT_HTTPHEADER,$header_arr);
```

此选项可以与上面提过的自定义头部内容结合。

7. CURLOPT_SSL_VERYFYPEER

此选项适合于目标服务器使用了SSL加密，即访问协议为HTTPS时用到。代码如下：

```
curl_setopt($ch,CURLOPT_SSL_VERYFYPEER,FALSE); //没有使用证书
```

8. CURLOPT_SSL_VERSION

此选项用来定义与目标服务器的 SSL 版本相对应，使用的是 SSL 版本（2 或 3）。默认情况下，PHP 会自己检测这个值，尽管有些情况下需要手动地进行设置。代码如下：

```
curl_setopt($curl, CURLOPT_SSLVERSION, 3);
```

特别是近两年的 SSL 出血漏洞，很多服务器又提高了安全等级，如连接微信的公众平台，腾讯已经废弃了 2 版和 3 版，改成了 TLS。因此，需要设置成如下代码：

```
curl_setopt($curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1);
```

9. CURLOPT_USERPWD 和 CURLOPT_UNRESTRICTED_AUTH

此选项适合于对方服务器有基本认证的情况（本书有单独的一章介绍此内容），这两个选项分别用于加入基本认证的用户名和密码。当访问存在基本认证时，每访问一个网页都提交一次用户名和密码，代码如下：

```
curl_setopt($ch,CURL_USERPWD," username:password" );
curl_setopt($ch,CURLOPT_UNRESTRICTED_AUTH,TRUE);
```

如果在整个 cURL 操作期间，我们还使用了 CURLOPT_FOLLOWLOCATION 选项，那么还应该配置 CURLOPT_UNRESTRICTED_AUTH 选项，以确保用户和密码会发送给所有重定向的页面（相同的域名）。

值得一提的是，使用此选项需要谨慎，如果给服务器发送了过期错误的用户名密码，会出现在目标服务器的访问日志中。

10. CURLOPT_POST 与 CURLOPT_POSTFILELDS

这两个选项用于通过 cURL 来模拟提交表单时的重要选项。默认值为 GET 方法，在提交处理前需要配置为 POST 方法，然后就可以向目标服务器发送 POST 数据了。请看如下代码：

```
curl_setopt($ch,CURLOPT_POST,TRUE); //设置为POST方法
$data = username=administrator&password=admin@admin.com; //定义POST的数据字符串
curl_setopt($ch,CURLOPT_POSTFIELD,$data);
```

POST 的数据似乎看起来很像 GET 方法里的查询字符串。一个小提示，如果使用 GET 方法提交表单信息的话，只需要在目标 URL 后加入查询的字符串就可以实现。

11. CURLOPT_VERBOSE

这个选项用来控制在文件传输过程中产生的状态信息数量。该选项在调试中会比较

实用。注意，如果在生产环境中使用，也会在服务器日志中留下记录。

可以关闭此详细模式，代码如下：

```
curl_setopt($ch,CURLOPT_VERBOSE,FALSE);  //日志最小化
```

12. CURLOPT_PORT

众所周知，在所有的 HTTP 会话中，默认采用 80 端口，cURL 默认值也为 80。比如在连接 SSL 加密的服务器时，它的端口变成 443，有的 HTTP 服务器为 8080 或 7001，不一而足。因此如果连接到自定义协议或非标准的 HTTP 服务器，就可使用 CURLOPT_PORT 来设置相应的端口号。代码如下：

```
curl_setopt($ch,CURLOPT_PORT,8080);  //使用端口8080
```

以上介绍了 cURL 最常用、也最核心的选项，需要注意，这里所示的配置选项必须为大写，也就是相当于 PHP 预定义的常量。如果换作小写，会产生错误。

接下来我们的实践会持续深入。首先介绍传输状态和错误信息的处理方法。

提示　本节主要介绍核心的参数，完整部分请参考 PHP 手册。完整的 cURL 选项网址：http://cn2.php.net/nanual/en/function.curl-setopt.php。

5.2.5　cURL 实践

1. 检测 cURL 错误与获取返回信息

我们常使用 cURL 模拟用户在浏览器访问网站，比如登录或提取网页中的内容、下载文件等操作。代码如下：

```
//初始化curl
$ch = curl_init() or die(curl_error());
//设置URL参数
curl_setopt($ch, CURLOPT_URL,"http://www.baidu.com/s?wd=php");
//要求curl返回数据
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
//执行请求
$result = curl_exec($ch) or die(curl_error());
//如果检查没有出错
if($result === FALSE){
    echocurl_error($ch);
}else{
    //取得返回的结果并显示
```

```
    echo $result;
}
//关闭curl
curl_close($ch);
```

上面的程序将从百度上的搜索结果页面获取并显示出来。我们这段代码和前面的脚本逻辑没有本质的变化，只是加了一段错误检测的过程。

```
if($result === FALSE){
    echocurl_error($ch);
}else{
    //取得返回的结果并显示
    echo $result;
}
```

在这里使用了“===”来进行严格比较，目的是区分空输出和布尔值 FALSE。

还有一个函数 curl_getinfo() 可以返回 cURL 执行后这一请求的相关信息，对于中间调试和排查错误很有用。

比如我们要获取 HTTP 返回的值，代码如下：

```
//etc
$http_code = curl_getinfo($ch, CURLINFO_HTTP_CODE);
echo 'HTTP code: ' . $http_code;
```

上面的脚本执行后，如果请求的 CURL 存在，会返回 200；如果不存在，即 404。

如果不设置参数，我们可以获取返回信息的所有内容的数组。代码如下：

```
//…
$curl_info = curl_getinfo($ch);
echo '<pre>';
var_dump($curl_info);
其返回的数组内容如下:
  ["url"]=>
  string(21) "http://www.21cto.com/"
  ["content_type"]=>
  string(24) "text/html; charset=UTF-8"
  ["http_code"]=>
  int(200)
  ["header_size"]=>
  int(991)
  ["request_size"]=>
  int(52)
  ["filetime"]=>
  int(-1)
```

```
["ssl_verify_result"]=>
int(0)
["redirect_count"]=>
int(0)
["total_time"]=>
float(0.080907)
["namelookup_time"]=>
float(2.4E-5)
["connect_time"]=>
float(2.4E-5)
["pretransfer_time"]=>
float(2.5E-5)
["size_upload"]=>
float(0)
["size_download"]=>
float(39744)
["speed_download"]=>
float(491230)
["speed_upload"]=>
float(0)
["download_content_length"]=>
float(-1)
["upload_content_length"]=>
float(0)
["starttransfer_time"]=>
float(0.078083)
["redirect_time"]=>
float(0)
["redirect_url"]=>
string(0) ""
["primary_ip"]=>
string(13) "115.28.174.91"
["certinfo"]=>
array(0) {}
```

类似于 $_SERVER 数组，这些值在调试 cURL 时让我们能够充分了解情况。比如抓取出错，就可以打印此数组中特定的键值来查看。比如在使用 cURL 抓取远端内容时，可能由于网络或对方服务器线路等问题，会出现数据下载不完整、超时等情况，这时候我们需要加一些文件校验。

我们可以通过获取的数据来计算文件尺寸（filesize），然后和 curl_getinfo 获取的数据进行比较，如果两者相等就认为下载正确，如果不是就重复尝试，如果几次都不成功，则放弃或加入失败队列，记入日志或过一段时间再行处理。这是在一种理想的情况下，

如果对方服务器返回有问题，返回的尺寸不全，两者对比也是相同的，但文件是不完整的，通俗来讲文件是坏的。

2. 抓取远程图片

我们编写的爬虫程序经常需要使用 cURL 来下载远程图片，抓取后再与远端返回的图片校检尺寸，以保证数据的完整性，如代码清单 5-16 所示：

代码清单5-16 cURL抓取远端文件

```
$source = "https://test.21cto.com/bookdownload/book.pdf";   //注意，网址是SSL，此
    处网址为虚拟
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $source);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($curl, CURLOPT_SSLVERSION, CURL_SSLVERSION_TLSv1);  //cURL设置选项为TLS
$data = curl_exec ($ch);
$info = curl_getinfo($ch);
curl_close ($ch);
$dest = "./files/test.pdf";
file_put_contents($dest,$data);
$size = filesize($dest);
if($info['size_download']!=$size ){
  echo '数据下载未完全';
}
```

在此例中我们使用了 file_put_contents 保存文件，使用了 curl_getinfo() 函数取得远端文件的大小来与本地下载的文件比较，确保了文件的完整性。

3. 使用 cURL 上传图片

上面我们介绍的是下载图片，那么这次是上传图片，其实本质上很相似。代码清单 5-17 是上传图片之脚本清单，请注意里面的注释：

代码清单5-17 使用cURL自动上传图片

```
<?php
    $target_url = 'http://21cto.com/curl/accept.php';  //抓取URL
    $file_path = 'sample.jpg';
    $post = array('extra_info' => '123456','file_contents'=>'@'.$realpath($file_
        path));  //构造上传文件数组
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL,$target_url);
    curl_setopt($ch, CURLOPT_SAFE_UPLOAD,true);  //请注意该选项
    curl_setopt($ch, CURLOPT_POST,1);
```

```
curl_setopt($ch, CURLOPT_POSTFIELDS, $post);
 curl_setopt($ch, CURLOPT_RETURNTRANSFER,1);
$result=curl_exec ($ch);
curl_close ($ch);
print_r($result);    //打印上传之结果
```

我们上传一个文件，只需要把文件路径做一个 POST 数组传过去。另外文件路径须是一个绝对路径，否则 PHP 找不到该文件，则无法成功提交。

为保证环境兼容性和代码的整洁，我们使用 realpath() 来引用当前的绝对路径。

需要注意的是，如果 PHP 版本小于 5.5，要在前面加上 @ 符号；如果你的 PHP 版本大于 5.5，则要使用新的上传构造语法。代码片断如下：

```
curl_setopt(ch, CURLOPT_POSTFIELDS, [
    'file' => new \CURLFile(realpath('sample.jpg')),
]);
```

CURLFile 方法是在 PHP5.5 中新增的，此时还兼容旧的语法，在 PHP5.6 中已彻底废除了 @ 语法。那么我们的代码要充分照顾到版本的兼容性，在文件字段处加上判断处理。代码片断如下：

```
if (class_exists('\CURLFile')) {
    $post = array('extra_info' => '123456', 'file_contents' => new \CURLFile
        (realpath($file_path)));
} else {
    $post = array('extra_info' => '123456','file_contents'=>'@'.$realpath($file_
        path));
}
```

即判断当前 PHP 虚拟机是否支持 CURFile，要根据环境来构造提交的文件路径和参数。另外，如果是 PHP5.5 以上版本，在 cURL 选项中需增加安全上传设置：

```
curl_setopt($ch, CURLOPT_SAFE_UPLOAD,true);
```

了解了核心功能，下面是我们完整地上传的代码清单，如代码清单 5-18 所示：

代码清单5-18　兼容的cURL自动上传功能

```
<?php
    $target_url = 'http://example.com/curl/accept.php';
    $file_path = 'sample.jpg';  //要提交的文件名
    //PHP版本兼容性处理
    if (class_exists('\CURLFile')) {
    $post = array('extra_info' => '123456', 'file_contents' => new \CURLFile
        (realpath($file_path)));
```

```
    } else {
    $post = array('extra_info' => '123456','file_contents'=>'@'.$realpath($file_
        path));
    }
    //开始执行cURL
$ch = curl_init();
      curl_setopt($ch, CURLOPT_URL,$target_url);
      curl_setopt($ch, CURLOPT_SAFE_UPLOAD,true);
      curl_setopt($ch, CURLOPT_POST,1);
      curl_setopt($ch, CURLOPT_POSTFIELDS, $post);
      curl_setopt($ch, CURLOPT_RETURNTRANSFER,1);
      $result=curl_exec ($ch);
      curl_close ($ch);
      print_r($result);
```

我们的文件是提到在accept.php上的，它用来接收文件和参数并返回信息。这个文件的代码如代码清单5-19所示：

代码清单5-19 接收文件和参数的脚本

```
<?php
    $upload_dir = realpath('./') . '/';
    $upload_file = $uploaddir . basename($_FILES['file_contents']['name']);
    echo '<pre>';
    if (move_uploaded_file($_FILES['file_contents']['tmp_name'], $upload_file))
    {
        echo "文件上传成功.\n";
    } else {
        echo "文件上传失败!\n";
    }
    echo '提交的文件内容:';
    print_r($_FILES);
    echo "\n<hr />\n";
    print_r($_POST);
    print "</pr" . "e>\n";
?>
```

可以看到接收文件端和正常的上传处理相同。

另外再说明一点，如果你用的是PHP5.5，CURL_SAFE_UPLOAD默认为FALSE，到了5.6后改成了TRUE。

从这点上看，PHP的新版本与旧版本过渡有一些小麻烦，需要各位多留意。

4. cURL批量处理

cURL还为我们提供了批处理的特性。这个功能允许开发者同时或异步打开多个

cURL 连接，如代码清单 5-20 所示：

代码清单5-20　使用curl_muilti函数并发执行cURL

```
<?php
  // cURL的URL地址和选项配置
  $ch_1 = curl_init('http://www.aa.com/');
  $ch_2 = curl_init('http://www.bb.com/');
  curl_setopt($ch_1, CURLOPT_RETURNTRANSFER, true);
  curl_setopt($ch_2, CURLOPT_RETURNTRANSFER, true);

  // 两个CUR资源再加入到$mh句柄中
  $mh = curl_multi_init();
  curl_multi_add_handle($mh, $ch_1);
  curl_multi_add_handle($mh, $ch_2);

  // 执行批处理等待全部完成
  $running = null;
  do {
    curl_multi_exec($mh, $running);
  } while ($running);

  // 待完成后，获取返回的内容
  $response_1 = curl_multi_getcontent($ch_1);
  $response_2 = curl_multi_getcontent($ch_2);
  echo "$response_1 $response_2";
  //关闭各个句柄
  curl_muilti_remove_handle($mh,ch_1);
  curl_muilti_remove_handle($mh,ch_2);
```

需要说明的是，如果使用 curl_multi，需要等待的时间更长一些。

在上面脚本中是打开多个 URL 句柄并指派给一个批处理句柄，然后在一个 do/while 循环里等它执行完毕。这个循环重复调用 curl_muilti_exec() 函数。这个函数是有阻塞（block）的，但它会尽可能减少执行。

另外有一个关于 curl_muilti 的无阻塞模式（non-block）的产品实现，我们可参考 URL：https://github.com/joshfraser/rolling-curl/。

5. 使用 cURL 模拟 Cookie 登录

Cookie 是保存在客户端的数据，用于与服务器端的数据或会话认证关联。比如，用户登录、订单兴趣信息等。

我们有时会想自动化登录网站或提交表单操作，尤其在一些会员登录账户中，需要发送 Cookie 才能实现登录，使用 cURL 是容易实现的。以下是对一个使用 get 方法的表

单的登录，如代码清单 5-21 所示：

代码清单5-21　使用cURL自动登录

```
//初始化cRUL
$ch = curl_init();
//设置登录表单的cURL地址
curl_setopt($ch, CURLOPT_URL, 'http://mis.example.com/index.php');
// 使用POST方法
curl_setopt ($ch, CURLOPT_POST, 1);
//设置POST的参数，表单名称和每个表单字段元素
curl_setopt ($ch, CURLOPT_POSTFIELDS, 'USERNAME=administrator&PASSWORD=sodopas
    s&imageField=&Action=Login');
//模仿浏览器的行为，并保存Cookie内容
curl_setopt ($ch, CURLOPT_COOKIEJAR, 'cookie.txt');
//设置CURLOPT_RETURNTRANSFER变量的值为1，表示curl_exec()方法返回一个字符串返回值
curl_setopt ($ch, CURLOPT_RETURNTRANSFER, 1);
// 开始执行请求(表单登录)
$store = curl_exec($ch);
//设置文件下载的参数
//curl_setopt($ch,CURLOPT_URL, 'http://mis.example.com/secret/file.php?file_
id=12');
//执行第二次请求（文件下载）
$content = curl_exec ($ch); //$content为取得的文件内容
//关闭cURL对象
curl_close ($ch);
```

这样，就可以使用脚本来自动登录到某个应用上，比如公司的办公系统，让它每天自动签到打卡，当然这仅是一个玩笑。

6. 使用 cURL 抓取 58 同城

在前面我们已经把 PHP cURL 功能的主要内容掌握了，因为 PHP cURL 函数非常灵活且具有这么多的配置参数，通常情况下我们可以封装一个函数来使用它比较方便，这样可以把复杂的代码更简化。

结合本节的内容，我们将 cURL 主要功能封装简化为一个函数，如代码清单 5-22 所示：

代码清单5-22　封装后的cURL函数

```
<?php
    function get_url_contents($url,$is_wechat='1'){
            if (function_exists('curl_init'))
            {
```

```
        $header = array();
        $aurl = parse_url($url);
        $header[] = 'Accept:text/html,application/xhtml+xml,
            application/xml;q=0.9,*/*;q=0.8';
        $header[] = 'Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,
            en;q=0.3';
        if($is_wechat == 1)
        {
                $header[] = "User-Agent: Mozilla/5.0 (iPhone; CPU
                    iPhone OS 6_1_2 like Mac OS X) AppleWebKit/536.26
                    (KHTML, like Gecko) Mobile/10B146 MicroMessenger/
                    5.0";
        }
        else
        {
                $header[] = 'User-Agent: Mozilla/5.0 (Windows NT 5.2;
                    rv:25.0) Gecko/20100101 Firefox/25.2';
        }
        $header[] = 'Host: ' . $aurl['host'];
        $header[] = 'Connection: Keep-Alive';
        $header[] = 'Cookie: tracknick=\u590F\u6587\u8F69';
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_FAILONERROR, false);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
        curl_setopt($ch, CURLOPT_POST, false);
        curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, 0);
        curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, 0);
        @curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
        @curl_setopt($ch, CURLOPT_MAXREDIRS, 10);
        @curl_setopt($ch, CURLOPT_HTTPHEADER, $header);
        $pageContent = @curl_exec($ch);
        $info = @curl_getinfo($ch);
        curl_close($ch);
        if ($pageContent && isset($info['http_code']) && $info
            ['http_code'] == '200')
                return $pageContent;
}else {
        if (extension_loaded("zlib")) {
                $pageContent = @file_get_contents("compress.zlib://" .
                    $url);
        }
        if (!$pageContent) {
                $pageContent = @file_get_contents($url);
        }
```

```
                return $pageContent;
            }
        }
    ?>
```

上面代码的主要功能是能够模拟微信浏览器访问目标服务器，并做了容错处理，如果服务器没有安装 cURL，将启用 file_get_contents() 函数的解决方案。我们主要看 cURL 的处理部分，相信你看过前面的内容，已经很熟悉这些函数和设置选项了，把该函数保存到 lib 目录下，命名为 functions.php。

首先明确功能目标，是 58 移动端网站 m.bj.58.com，注意这个网站只能使用微信客户端访问。我们之所以选择移动端 HTML5 网站，主要的原因是页面结构简单，抓取速度更快。

如果想用 PC 浏览器访问移动端网站，可以在 chrome 浏览器上安装 agent 代理插件如 User-Agent Switcher for Chrome，然后选择 iPhone 或 Android 设备就可以正常浏览了，如图 5-15 所示。

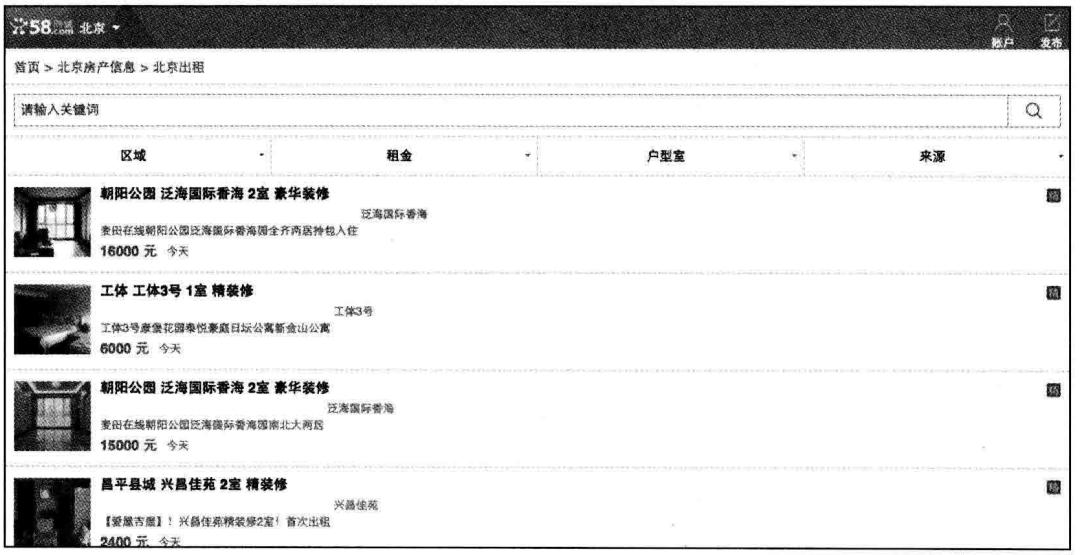


图 5-15　58 同城房源列表页面

我们看到这是一个房源列表的页面，并且有很多分页。那么我们的爬虫原理是将每个列表页遍历一次，找到房源详细的链接，然后进入该页面抓取需要的数据，再进入下一个列表页，依次类推。

下面先取得每个列表页房源详细页的链接。代码如下：

```
$file_content = get_url_contents($url);
//房源列表内容
preg_match_all("/<ul  class=\"list-info\">(.*)<\/ul>/isU",$file_content,  $house_
    lists);
//print_r($house_lists);exit;

//$content=preg_replace("/[\t\n\r]+/","",$content);
echo '当前URL地址: '.$url.'<hr>';

foreach($house_lists[0] as $house_link)
{
   preg_match_all('/<a href="(.*?)\"/s', $house_link, $house_item);
}
```

图 5-16 所示是详细页面的画面，主要是房源的标题和参数以及详细内容，我们要的关键内容都在这里了。

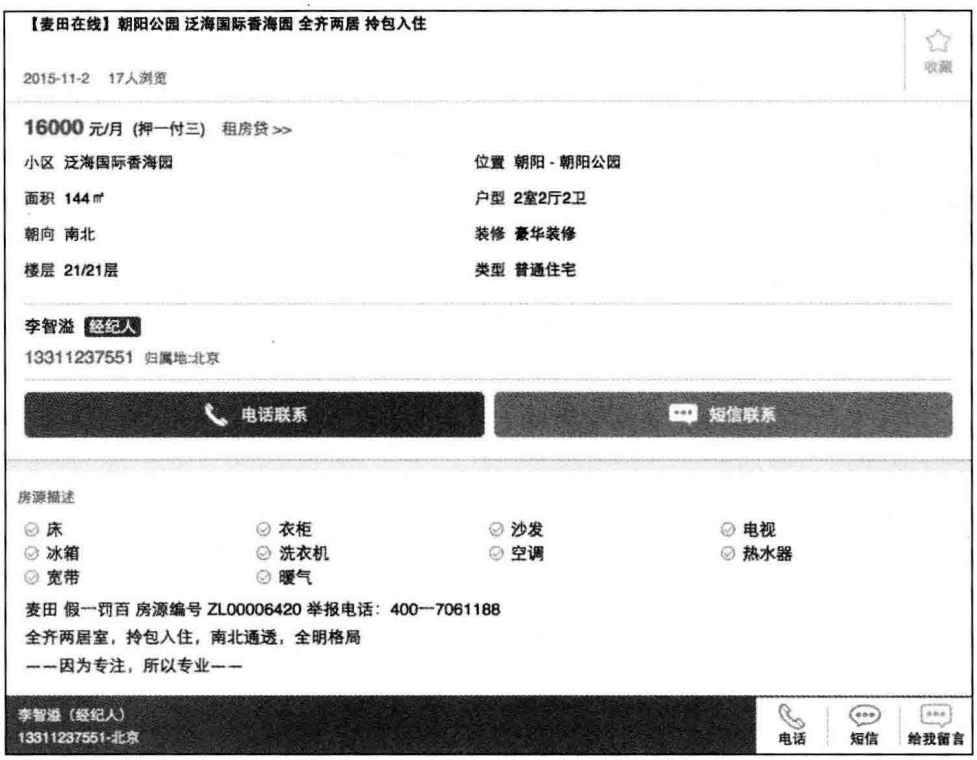


图 5-16　房源列表详细页面

下面对此页面进行分析，这里我们主要取的是 HTML 的关键标签“ fang_new”中间的部分，并使用了正则表达式获取。代码如下：

```
//房屋字段内容
preg_match_all("/<ul class=\"attr_info fang fang_new\">(.*)<\/ul>/siU", $house_
    details, $house_info);
//联系人，电话
preg_match_all("/<p class=\"llname\">(.*)<\/p>/siU", $house_details, $contact_
    personal);
preg_match_all("/<p class=\"llnumber\">(.*)<\/p>/siU", $house_details, $contact_
    number);
//图片
preg_match_all("/<div class=\"image_area
    image_area_new\">(.*)<\/div>/siU", $house_details, $house_images);
preg_match_all('/ref="(.*?)\"/s',$house_images[1][0], $house_image);

echo '房源图片: ';
foreach($house_image[1] as $img)
{
   echo $img ."<br />";
}
```

通过正则表达式的关键标签匹配，我们拿到了房源信息，包括名称、介绍、联系信息（电话、地址、房型）、房型图片等内容，抓取过来后放在本地的数据库或文件中。

以下为抓取的完整代码，如代码清单 5-23 所示：

代码清单5-23　抓取的完整代码

```
<?php
   include('libs/functions.php');
   $page = $_GET['pg'];          //通过页面传值
   $url = 'http://m.bj.58.com/chuzu/'.$page;

   $file_content = get_url_contents($url);
   //房源列表内容
   preg_match_all("/<ul class=\"list-info\">(.*)<\/ul>/isU",$file_content, $house_
       lists);
   //print_r($house_lists);exit;

   //$content=preg_replace("/[\t\n\r]+/","",$content);
   echo '当前URL地址: '.$url.'<hr>';

   foreach($house_lists[0] as $house_link)
   {
          preg_match_all('/<a href="(.*?)\"/s', $house_link, $house_item);
```

```
}

foreach($house_item[1] as $item_url)
{
        echo '详细页URL='.$url.'<br />';

        $house_details = get_url_contents($item_url);
        if(!empty($house_details))
        {
                //print_r($house_details);
                //房屋字段内容
                preg_match_all("/<ul class=\"attr_info fang fang_new\">(.*)
                    <\/ul>/siU", $house_details, $house_info);
                //联系人，电话
                preg_match_all("/<p class=\"llname\">(.*)<\/p>/siU", $house_
                    details, $contact_personal);
                preg_match_all("/<p class=\"llnumber\">(.*)<\/p>/siU",
                    $house_details, $contact_number);
                //图片
                preg_match_all("/<div class=\"image_area image_area_new\">
                    (.*)<\/div>/siU",$house_details, $house_images);
                preg_match_all('/ref="(.*?)\"/s',$house_images[1][0], $house_
                    image);

                echo '房源图片: ';
                foreach($house_image[1] as $img)
                {
                        echo $img ."<br />";
                }
                //远程复制图片

                print_r($house_info[1]);
                //以及房源信息，注意入库前去重
                print_r($contact_personal);
                print_r($contact_number);
        }
        set_time_limit(1000);
        //exit;
}
```

这个脚本还需要和一个静态页面配合使用，静态页面中设置抓取的页码，在浏览器里循环进行抓取，同时保证与对方网站会话的持续。如果目标服务器没有会话控制，也可以使用 CLI 方式处理。

代码清单 5-24 是静态页面的代码清单，它依赖 JQuery 库。内容如下：

代码清单5-24 抓取前端页面

```
<!doctype html>
<html>
<head>
<meta charset="utf-8">
<title>抓取58数据</title>
<!—使用jquery库-->
<script src="assets/js/jquery-1.11.0.min.js"></script>
<script>
var start_num = 0;
var end_num = 2;  //抓取的最大页码数

function loopLi()
{
    setInterval(function(){
        if (start_num < end_num) {
                $.ajax({
                        type: 'GET',
                        url : "get-58-data.php",
                        data: { pg : start_num },
                        cache : false,
                        crossDomain: false,
                        success: function(data)
                        {
                                $('#result_info').html('<li>'+data+'</li>');

                        },
                        error : function(XMLHttpRequest, textStatus,
                            errorThrown) {
                                //alert("Error occured while loading Loads."+
                                    textStatus);
                        }
                });
                start_num++;

        }

    }, 500);
}

$(loopLi);
</script>
</head>

<body>
```

```
<div name='result_info' id='result_info'>
</div>
</body>
</html>
```

抓取工具的挑战之处还有一个，就是如果目标网站修改了页面，我们需要与时俱进式地调整匹配规则。比如你在阅读本节时，58 的前端工程师或许已经修改了页面，上面的正则表达式需要重新调整。所以在一些客户端抓取工具中，我们也可参考一些规则，可以节省分析页面的时间。

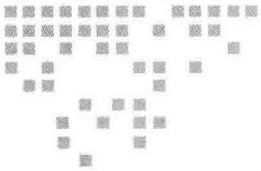
当然如果对方与我们是合作关系，开发 API 是最好的数据打通方式，这样 cURL 就成了正常的 API 接口。另外我们还会在专门的著作中详细讲述爬虫的开发。

5.3　本章小结

我们从介绍 Socket 的基础开始，到如何使用 PHP 来开发 Socket 服务器端与客户端。另外也讲解了使用 cURL 来抓取网页数据，这是非常重要的技能。例如很多的开发平台，如微信、新浪微博、淘宝等，都可以用 cURL 处理。虽然 cURL 的底层也是 Socket，对于开发者来讲，使用 cURL 库会更方便，这取决于你的选择。

大多数搜索引擎和推荐引擎，如 Google、百度、今日头条等产品的前期工作均在数据抓取上做了很多事，希望大家多理解并延伸实践。

当我们综合运用这些技术来获取、分析、处理这些海量信息时，会发现自己掌握这么强大的工具，收获成就感之同时，也许很快就成为研发团队中的核心人物了。



Chapter 6 第 6 章

PHP 调优、测试与工具

无论怎样深思熟虑，程序总会难免有这样或那样的性能或逻辑问题。可以确定的是，这些事总是会被我们解决，性能得到有效提升，逻辑回归正确。但是如果有合适的工具帮助我们快速定位问题，就会使工作事半功倍，避免这些很耗费精力的事。我们都是人，人在开发中都会犯些错误，现实情况中也少有完美的程序员。

性能分析是对应用程序代码级别的表现捕获，比如 CPU、内存使用率，函数调用数和执行时间，生成调用图形，分析哪些行为导致性能问题。

常见的原因有很多种，包括：数据库端如 MySQL、PostgreSQL、MongoDB；内存数据库如 Redis、Memcache；外部问题造成的问题，如磁盘满或损坏、网络带宽、代码问题等。

根据一些不严谨统计数据，每个软件项目中的代码平均有 7 行以上错误。除去“写一行，导致 BUG 千行”这种泪奔错误外，貌似有些天文数字，夸张中也带点儿心酸味道。在现实中除了我们粗心大意之外，更多的是语法错误或不正确的逻辑。因此每一位开发者都要用一套工具包，积累开发中的调试经验，应对形形色色的大小 BUG，小到语法错误，大到大段代码重构。善于调试代码，这是平庸程序员和优秀工程师的差别所在。

下面我们就来看如何调试与调优 PHP。

6.1　PHP 调试

事实上 PHP 和 MySQL 都提供了一些函数库供我们来诊断与调试运行中的程序，另外我们还可以借助第三方工具来进行增益。对于正常的应用程序，使用基本的调试工具就可以了，最常见的是使用 print_r() 等函数，或者使用循环来显示数组或类的内容，比如使用 var_dump() 比 print_r() 会得到更多的信息。但对付更复杂的应用，有时想检查应用程序的更深入的状态和性能，靠基本调试语句就稍显力不从心了。

在本节，我们就讲解如何了解代码运行的状态及程序运行时的性能调试。

6.2　语法检查

PHP 解释器在执行脚本时的第一件事就是检查语法，以确保基本语法正确，否则立即抛出错误，中止执行。

为此，我把常见的错误归纳为以下几点：

- 一行语句中没有用分号结束声明。
- 引用函数时，存在同名函数。
- 代码结构不完整。

当以上这些问题发生时，PHP 解释器往往提供的信息并不是很详细，因此我们需要花一些心思找到问题所在，然后才得以解决。代码清单 6-1 所示是一个最简单的 PHP 脚本，代码如下：

代码清单6-1　一个最简单的PHP脚本

```
<?php
echo "Hello World!'
?>
```

注意，这个脚本是含有错误的，即在表达式的结尾没有使用分号结束，但是我们很多时候没有发现，就会出现错误，如果测试环境没有问题，如果是面向用户的网站，则不能显示任何错误信息给用户。

因此，我们写好一个程序后，可以使用 PHP 命令行来检查脚本中的语法，格式为：

```
php -l  <脚本文件>
```

该命令在 Windows 系统或 Linux 系统都可以使用，如图 6-1 所示。

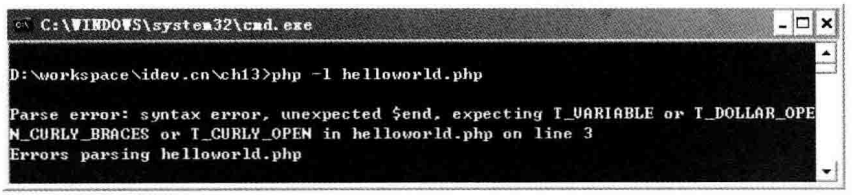


图 6-1 用 PHP 命令行检查脚本语法

可以看到，PHP 解析器发现了错误，提前终止了执行。但是它提示我们第 3 行出错，而实际上的原因是我们没有在 echo 语句后加分号，另外字符定界符不匹配，开始使用的是双引号，后面使用了单引号。

本节中使用的 PHP 版本是 5.3.2，如果你使用的版本在 5.3 之前，错误提示可能会不一样，请注意鉴别。

我们把脚本中的错误修正好后再运行一次，如图 6-2 所示。

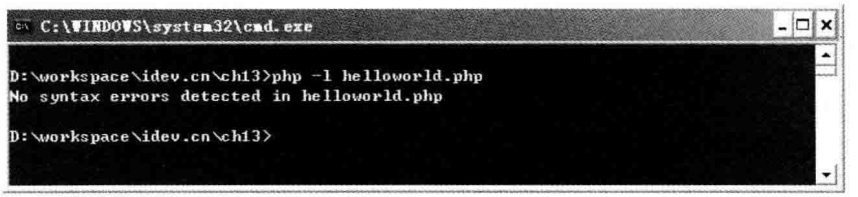


图 6-2 修正后的运行结果

可以看到 PHP 输出了信息：No syntax errors detected in helloworld.php，说明该脚本在语法上没有错误。

这证明它能做一些事情，但是脚本能不能完成任务，这是另外一个问题。虽然我们的代码通过了语法检查，但这并不表示脚本里没有其他逻辑错误，也就是业务应用的问题，PHP 并不知道我们的脚本具体干什么，也无法做到智能纠正。因此还要一些更好的工具来帮助我们看到执行中的逻辑。

6.3 输出调试信息

前面我们一直关注在 PHP 中生成错误的信息，接下来我们一起来思考如何实时追踪到一些信息，以便能知道程序的应用逻辑是否正确，以能够快速定位到错误。

我将在下面讨论如何利用 PHP 提供的一系列内置函数和常量，然后尝试把它们抽象成一个类，以便在调试代码时给我们更多的方便和详细信息。

6.3.1　使用内部函数调试

有几个核心的 PHP 函数对于输出和格式化调试信息非常有用，有一部分你可能已经很熟悉。下面列出这些函数，可以选择最适合自己的来用。

使用 echo 或 print 是最简便的调试方法，一般用来输出变量值，或者你不确定程序执行到了哪个分支的情况下使用，如代码清单 6-2 所示：

代码清单6-2　一个特殊的分支语句

```
<?php
$var = 2;
if(isset($var)){
    echo 1;
}elseif(isset($var1)){
    echo 2;
}elseif(isset($var2)){
    echo 3;
}
?>
```

对于简单的变量清楚地看到代码分支执行到了第一个判断分支。

这种在浏览器上的输出方式，没有比 echo 或 print() 函数更简单的了，使用哪一个取决于你自己的喜好。echo 执行稍快一点儿，print() 函数会有一个返回值，虽然两个函数差异与我们现在的目的无关，不过使用 echo 还是比 print() 函数更常用。

var_dump 和 print_r 两个函数可以输出结构信息，如果参数是一个数组或对象，它们将遍历层次结构和显示有关信息。这两个函数可以打印变量，但主要是针对特数组和对象数据，一般我们在查看接口返回值，或者某些不太确定变量的时候，都可以使用这两个 API。

最简单常见的调试方法，一般就是调用 print_r 或 var_dump 之类的输出 API，直接在浏览器中输出内 容，查看内容来进行调试工作，如代码清单 6-3 所示：

代码清单6-3　常见的调试方法

```
<?php
$arr1 = array(
    "test1" => "val1",
    "test2" => "val2",
);

echo "<h2>浏览器调试</h2>";
```

```
echo "<pre>";
print_r($arr1);
echo "</pre>";
```

浏览器输出结果如图 6-3 所示。

图 6-3 浏览器输出结果

我们来看这两个函数的一些主要区别：

一般来说，print_r() 函数输出的内容更易于阅读。它还可使用第 2 个可选参数来返回布尔值，不是只输出到浏览器。我们也可从 var_dump() 函数取得类似的结果，还可以使用 PHP 的输出控制和输出缓冲函数 [ob_start() 和 ob_end_flush()] 来控制输出。

另一个区别是，var_dump() 函数输出的信息会多一些，比如它会告诉你一个变量或属性的数据类型以及它的大小。下面我们写一个类，该类的主要功能是将调试信息写入 CSV 或纯文本文件中，如代码清单 6-4 所示：

代码清单6-4 调试时写入日志

```
<?php
class MyDump{
    //私有的、保护的和公共属性
    private $path = '/this/way';
    protected $extensions = array('csv', 'txt');
    public $file = null;
    //构造方法
    public function __construct($file){
            $this->$file;
    }

    //私有方法
```

```
    private function getPath(){
            return $this->path;
    }

    //保护的方法
    protected function getFile(){
            return $this->file;
    }
    //公共方法
    public function getPathAndFile(){
            return $this->getPath() . '/' . $this->getFile();
    }
}

//初始化对象
$mdump = new MyDump('data.csv');
//使用var_dump
var_dump($mdump);
//使用print_r
//echo print_r($mdump, TRUE);    //可以打开该注释进行对比
print_r($mdump);
```

图 6-4 所示是该脚本执行的结果：

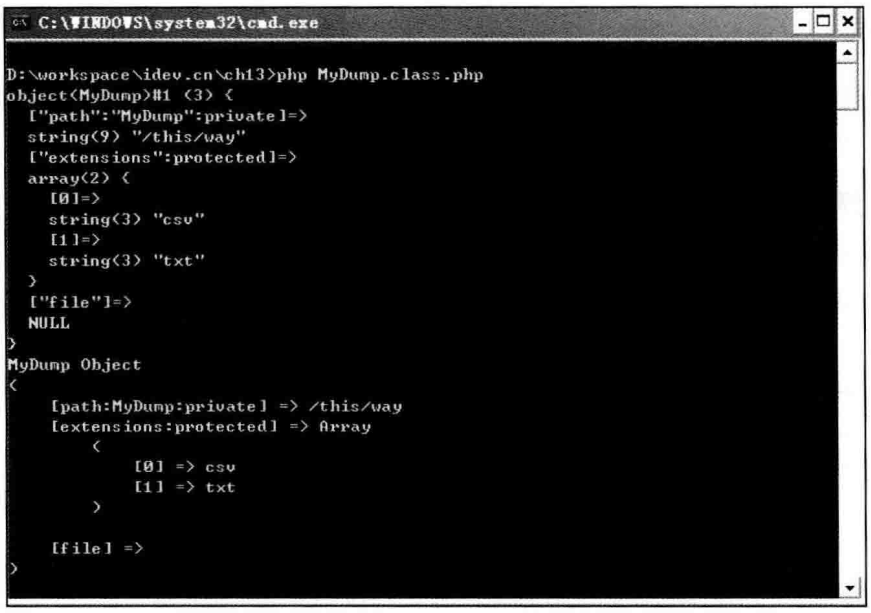


图 6-4　脚本执行的结果

我们看到，print_r() 函数输出的结果比较简洁，而 var_dump() 函数输出的结果则比较详细。另外，比较有趣的是，两个函数都能访问类的私有变量和内容，却没有显示类中方法的信息。

下面我们继续了解 hightlight_string() 和 highlight_file() 这两个函数。highlight_string() 函数会将传它的第 1 个参数以高亮的方式输出，highlight_file() 函数也做同样的事情，只是会将一个文件的内容以高亮方式输出。

我们可以以下面的表达式来理解这两个函数的关系：

```
highlight_string (file_get_contents(filename)) == highlight_file(filename)
```

如果两个函数都给了第 2 个参数为真，它们将返回值为 1，而不是输出高亮字符串；如果没有输出成功则返回假，表示输出失败。

这两个函数对于在代码中明显标注错误信息是很有用的，但是在正式的产品上要慎重使用这两个函数，因为会存在一个安全风险，一旦输出到浏览器端，里面的信息将暴露给网站所有的用户，包括攻击者与黑客。

在调试 PHP 程序时，通常在执行流程设置一个断点，程序到此中止执行，输出相应的调试信息。大多数情况下，我们调用 exit() 函数即可以中止执行，也可以使用 exit(' 调试信息 '); 来输出自己的调试信息，如 exit(" 结果是：$result\n");，将输出 $result 变量的值，然后再退出程序执行。

另外 exit() 函数还可以使用整数作为参数，它将返回一个整数后终止执行。整数 0 表示程序正常终止，而任何大于 0 的整数则表示一个错误条件。

使用魔术常量

所谓魔术常量，就是这个常量保存着当前 PHP 脚本运行时的状态，如当前脚本名称、运行的行号等内容，见表 6-1，我们可以根据需要引用这些常量值。

表 6-1 PHP 的“魔术常量”列表

名　称	说　明
LINE	返回运行中 PHP 脚本的当前行号
FILE	返回当前执行 PHP 脚本的完整路径和文件名，包含一个绝对路径
FUNCTION	函数名称，返回该函数被定义时的名字
CLASS	类名称，返回该类被定义时的名字
METHOD	类的成员方法名称，返回该方法被定义时的名字（大小写敏感）
DIR	目录，返回当前脚本的目录

（续）

名　称	说　明
NAMESPACE	命名空间，返回当前脚本的命名空间

6.3.2　建立堆栈跟踪

前面的函数均为基本的调试函数，下面向大家介绍一些不常用但也很有用的函数。

1. debug_zval_dump()

debug_zval_dump() 输出结果跟 var_dump() 类似，唯一增加的一个值是 refcount，就是记录一个变量被引用了多少次，这是 PHP 的 copy on write（写时复制）的机制的一个重要特点。

```
<?php
$a = $b = $c = $d = "abc";
debug_zval_dump($a);
$x = $y = $z = $w = null;
debug_zval_dump($x);
$q = null;
debug_zval_dump($q);
?>
```

代码输出结果如下：

```
string(3) "abc" refcount(5)
NULL refcount(5)
NULL refcount(2)
```

我们仔细看代码，尽管变量 $ x 和 q 都是 NULL，它们是不一样的 zval。但 $ x 和 $ y 共享相同的 zval，因为它们是彼此分配的。

这个函数和 var_dump() 很类似，但唯一不同的是增加一个变量指针 refcount，记录一个变量被引用多少次。这是 PHP 写时复制的一个重要特点，和 PHP 变量在底层的实现及内存回收有关。

关于 debug_zval_dump() 函数的详细内容和关于 refcount 的详细内容在 http://php.net/manual/en/function.debug-zval-dump.php。

2. debug_backtrace()

使用 PHP 调试可以使用 debug_tacktrace() 函数，它会生成一个关联数组，数组里会收集当前应用程序的状态信息，另外还提供当前应用程序的堆栈信息以及堆栈中存储应

用程序的方法。

PHP 脚本运行时，每一个函数调用期间，无论是压栈还是出栈，当发现堆栈溢出（overflow）时，比如使用多层的递归调用 debug_backtrave()，函数返回当前的堆栈结果，并且返回当前程序以前的调用状态。

debug_backtrace 提供了一个非常详尽的信息，包括堆栈中代码行号和文件名，这些对于程序员来说是很有用的跟踪数据。

使用这个工具，只需要调用 debug_backtrace() 函数即可。我们使用如下格式，例如：

```
$debug = debug_backtrace();
```

该函数返回值为一个关联数组。该数组的第一个元素是当前堆栈的状态，之后是其他当前堆栈的状态。其中最为有用的是当例外发生时，会引发 back-trace 异常，并且会记录错误日志。

该数组元素依赖于调用它的当前代码的上下文。包括：

- 在 global 全局范围。
- 在包含文件内。
- 在函数调用时。
- 在类中一个方法被调用。

根据不同的范围和内容，一个或更多的以下键名（key）会记录如下内容：

- function 包括一个函数名称的字符串，它是唯一的，如果当前函数被调用。
- args 以一个数组形式来保存当前函数调用时的参数。若当前过程为调用内部函数，刚 args 不存在。

在下列文件包含的情况下，你将得到一个参数，这个函数名使用的包含文件。

- class 包括类的名称（如果有）。如果是一个类或者是一个对象（即类的实例引用），我们在调一个静态方法时，不会引用对象。
- type 一个更可靠的测试方法，如果一个方法被静态调用或使用对象调用，这个 type 变量的值是“ ->”，表示正常的调用，或“ : ”静态调用。如果当前过程不存在类调用，它将不被提供。
- file 和 line，包含当前文件和行号，该键名会一直存在。

值得注意的是，每个键值的范围是相对于堆栈点的当前范围而言的。回溯对于产品化，使用自定义日志甚至可以自动修复错误。

因为 debug_backtrace() 返回的是关联数组，我们可以将其封装为一个函数，这样可

以方便调用，并且对其中的内容格式化，代码如下：

```
<?php
function get_backtrace(){
    $history = debug_backtrace();
    $msg = '';

    foreach ($history as $i => $line) {
        $msg .= "#$i {$line['function']}() called at [{$line['file']}:{$line
            ['line']}]\n" ;
     }

    return $msg;
}


function a() {
    b();
}

function b(){
    c();
}

function c(){
    print_r(get_backtrace());
}

a();
```

该脚本的执行结果如下：

```
#0 get_backtrace() called at [D:\workspace\idev.cn\ch13\debug_trace.php:27]
#1 c() called at [D:\workspace\idev.cn\ch13\debug_trace.php:22]
#2 b() called at [D:\workspace\idev.cn\ch13\debug_trace.php:17]
#3 a() called at [D:\workspace\idev.cn\ch13\debug_trace.php:30]
```

我们可以清楚地看出 get_backtrace() 被调用的顺序和它所在的代码行。

3. debug_print_backtrace()

这个函数的名字和 debug_backstrace() 函数又有点儿像，debug_print_backtrace() 函数提供了比 debug_backtrace() 函数更多的细节。类似于刚才我们写的 get_backtrace() 函数，它会输出调用的函数名和行号等，这样我们前面的函数就可以不用了。来看如下代码：

```
<?php
function a() {
    b();
}

function b(){
    c();
}

function c(){
    debug_print_backtrace();
}

a();
```

该脚本的执行结果如下：

```
#0  c() called at [D:\workspace\idev.cn\ch13\debug_print_backtrace.php:9]
#1  b() called at [D:\workspace\idev.cn\ch13\debug_print_backtrace.php:4]
#2  a() called at [D:\workspace\idev.cn\ch13\debug_print_backtrace.php:17]
```

6.4 活用日志

日志（Log）记录也是很重要的一种调试和监控手段，一般的原则要求是尽量多地输出日志，在查找问题的时候比较容易定位，特别是线上业务，没有记录日志，要找问题是不可想象的。

日志记录除了PHP解析级别的错误，更多是程序在执行过程中的一些错误，比如文件资源打开错误（文件不存在、没有权限、文件格式不正确）、远程服务资源访问失败（网络不通、用户名密码错误）等，要知道，任何你认为不会出错的地方都可能隐藏着错误，所以务必尽量多地输出Log。

几个常用的文件操作API都可以完成日志的常用操作，比如fopen/fwrite，或者是一步到位的file_put_contents()。另外，为了便于写日志，PHP还为我们提供了一个专门的接口：error_log。

为了节约磁盘IO操作，可以放到一个对象里，等对象析构的时候再执行物理写入操作。下面我们来看一个简单的日志记录类：

```
<?php
class myLog{
   private $str = '';
```

```
    const LOG_LEVEL_ERROR = 1;
    const LOG_LEVEL_WARNING = 2;
    const LOG_LEVEL_NOTICE = 3;
    const LOG_FILE = "PHP_Log_%s.log";

    function __construct(){}
    function __destruct(){
        if$this->str != ''){
                $file = sprintf(self::LOG_FILE,date("Ymd"));
                file_put_contents($file,$this->str,FILE_APPEND);
        }
    }

    function log($str,$level){
        switch($level){
              case self::LOG_LEVEL_NOTICE:
                      $this->str .  "[".date("Y-m-d H:i:s")."] Notice:".$str."\n";
                      break;
              case self::LOG_LEVEL_WARNING:
                      $this->str .  "[".date("Y-m-d H:i:s")."] Warning:".$str."\n";
                      break;
             case self::LOG_LEVEL_ERROR:
                      $this->str .  "[".date("Y-m-d H:i:s")."] Error:".$str."\n";
                      break;
       }
      }

    function notice($str){
        $this->log($str,self::LOG_LEVEL_NOTICE);
    }

    function warn($str){
        $this->log($str,self::LOG_LEVEL_WARNING);
    }

    function error($str){
        $this->log($str,self::LOG_LEVEL_ERROR);
    }
}
```

我们测试输出 10 次日志：

```
for($i=0;$i<10;$i++){
    $log->notice("test $i");
    sleep(1);
}
```

直到脚本结束，你才会看到日志内容输出。因为 PHP 只有在强制调用析构函数时才会析构：

```
$log->__destruct();
unset($log);
```

一般情况下都是在脚本执行完毕的时候，才去调用每个对象的析构函数进行资源回收处理。

刚才提到过，我们除了使用 fopen/fwrite 和 file_put_contents 这两种文件操作来进行日志记录外，也可以使用 error_log 来进行记录。记录的类型，可以记录到系统的 syslogd 进程，也可以发送邮件，发送到远程日志记录服务器或记录到文件。

6.5 Xdebug

到目前为止，我们一直在讨论在脚本中如何单步调试和输出信息，现在我们将开始利用一个专业工具，将更有助于调试工作。

Xdebug（http://www.xdebug.org/）是一个 Zend 扩展包，能够使用钩子（Hook）到 PHP 的内部编译器，凭借这种能力，它添加了输出调试信息等一些特性。

在本节，我们要着重描述用于发现和诊断并修复错误的特性。特别是将研究提高性能的堆栈跟踪、函数和变量追踪，以及交互式脚本调试、本地和远程支持。

主要包括以下功能：

（1）栈和函数错误信息的追踪：

- 显示用户定义方法或函数的参数；
- 函数名称，文件名和行标记；
- 支持显示成员方法。

（2）内存分配情况。

（3）保护无限递归。

（4）用于 PHP 脚本信息分析。

（5）脚本执行分析。

（6）交互式脚本调试。

Xdebug 是一个开源项目，该项目主要开发和维护者是 Derick Rethans，他在 2003 年左右开发完成 Xdebug 并推出，并继续在此基础上开发升级。Xdebug 可以运行在 Macintosh、Linux 和 Windows 多个平台上，想了解更多信息可以查阅它的官方网站，如

图 6-5 所示。

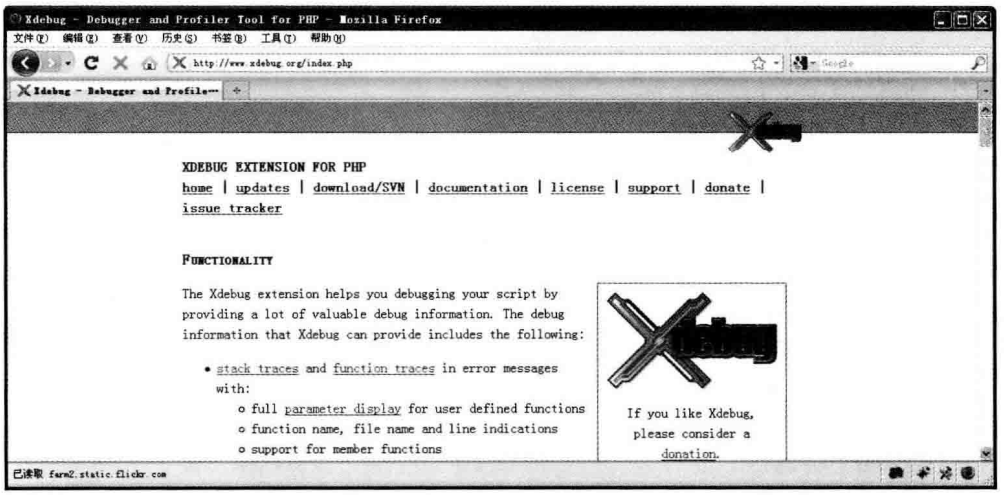


图 6-5　Xdebug 官方网站

安装 XDebug 的方法如下所述。

6.5.1　安装 Xdebug

1. 在 Linux 平台上安装 Xdebug

在 Linux 平台上安装 Xdebug 的首选方法是通过 PECL，我们使用 pear 命令安装 Xdebug：

```
# sudo pecl install xdebug
```

此时等待 PECL 安装完成，然后在 php.ini 文件下增加命令如下行：

```
zend_extension="/usr/local/php/modules/xdebug.so"
```

重启 Apache，打开 phpinfo() 页面，应该能看到 Xdebug 在版权页的输出，如图 6-6 所示。

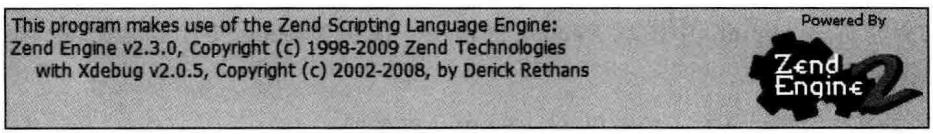


图 6-6　安装成功的 Xdebug

接着再往下移动页面，我们会看到 Xdebug 的配置信息，如图 6-7 所示。

xdebug support	enabled
Version	2.0.5

Supported protocols	Revision
DBGp - Common DeBuGger Protocol	$Revision: 1.125.2.6 $
GDB - GNU Debugger protocol	$Revision: 1.87 $
PHP3 - PHP 3 Debugger protocol	$Revision: 1.22 $

Directive	Local Value	Master Value
xdebug.auto_trace	Off	Off
xdebug.collect_includes	On	On
xdebug.collect_params	0	0
xdebug.collect_return	Off	Off
xdebug.collect_vars	Off	Off
xdebug.default_enable	On	On
xdebug.dump.COOKIE	*no value*	*no value*
xdebug.dump.ENV	*no value*	*no value*
xdebug.dump.FILES	*no value*	*no value*
xdebug.dump.GET	*no value*	*no value*
xdebug.dump.POST	*no value*	*no value*
xdebug.dump.REQUEST	*no value*	*no value*
xdebug.dump.SERVER	*no value*	*no value*
xdebug.dump.SESSION	*no value*	*no value*
xdebug.dump_globals	On	On
xdebug.dump_once	On	On
xdebug.dump_undefined	Off	Off
xdebug.extended_info	On	On
xdebug.idekey	dujiang	*no value*
xdebug.manual_url	http://www.php.net	http://www.php.net
xdebug.max_nesting_level	100	100
xdebug.profiler_aggregate	Off	Off

图 6-7 安装成功的 Xdebug

注意 在安装 Xdebug 时，就不能再使用其他任何 Zend 扩展库，尤其是和它类似的调试器，否则它们有可能发生冲突。

如果在你的 phpinfo 中发现 Xdebug 的输出，说明 Xdebug 没有正确安装。如果是，请注意以下几点：

- 确认安装完 Xdebug 后重启 Apache Web 服务器。
- 检查你的 php.ini 文件中是否包含调入 Xdebug.so 的行。在 Linux 下可以使用 tail 命令进行查找：

```
$tail -37 php.ini|more。
```

- 确定在上面指定的路径的文件存在。
- 查看 Web 服务器和 PHP 的日志文件。

由于 Xdebug 是一个 Zend 扩展，相关配置都可以写在 php.ini 文件中，因此 Xdebug 的默认配置会被 php.ini 中的参数覆盖。

此外，也可使用 .htaccess 文件或使用 PHP 脚本中的 ini_set() 来修改 Xdebug 的行为和控制它的输出。

以下是我的 php.ini 配置文件中设置的 Xdebug 段，代码如下：

```
[xdebug]
; tell PHP where to find the Xdebug extension and load it
zend_extension=/usr/local/apache2/php/lib/php/extensions/no-debug-non-zts-20060613/
    Xdebug.so
; protection
xdebug.max_nesting_level=100
; what to show when outputting variables & debug info
xdebug.show_local_vars=1
xdebug.collect_params=1
xdebug.var_display_max_children=128
xdebug.var_display_max_data=1024

xdebug.var_display_max_depth=5
xdebug.dump.COOKIE=*
xdebug.dump.FILES=*
xdebug.dump.GET=*
xdebug.dump.POST=*
xdebug.dump.REQUEST=*
xdebug.dump.REQUEST=*
xdebug.dump.SESSION=*
; enable & configure remote debugging
xdebug.remote_enable=1
xdebug.remote_host=127.0.0.1
xdebug.remote_port=9000
xdebug.remote_port=Xdebug.remote_handler=dbgp
xdebug.remote_mode=req
xdebug.remote_log=/tmp/Xdebug_remote.log
xdebug.remote_autostart=0
xdebug.idekey=dirk
; configure profiler - disabled by default, but can be triggered
xdebug.profiler_enable=0
xdebug.profiler_enable_trigger=1
```

```
xdebug.profiler_output_dir=/tmp
xdebug.profiler_output_name=Xdebug.out.%s
```

注意 大家知道，PHP 可以有多个 .ini 配置文件。通常这取决于它的 SAPI。例如，如果你正在使用 PHP 命令行，即使用的是 CLI SAPI 模式，将使用配置文件如 cli.ini。如果 Xdebug 安装后没有如期出现，你也可以找到相关正确的 .ini 文件。

2. 在 FreeBSD 平台上安装 Xdebug

接下来再向大家介绍在 FreeBSD 平台安装 Xdebug 的方法。代码如下：

```
#cd /usr/ports/devel/php-xdebug
#make install
```

我们等待它安装完成后，最后将提示类似如图 6-8 所示的信息。

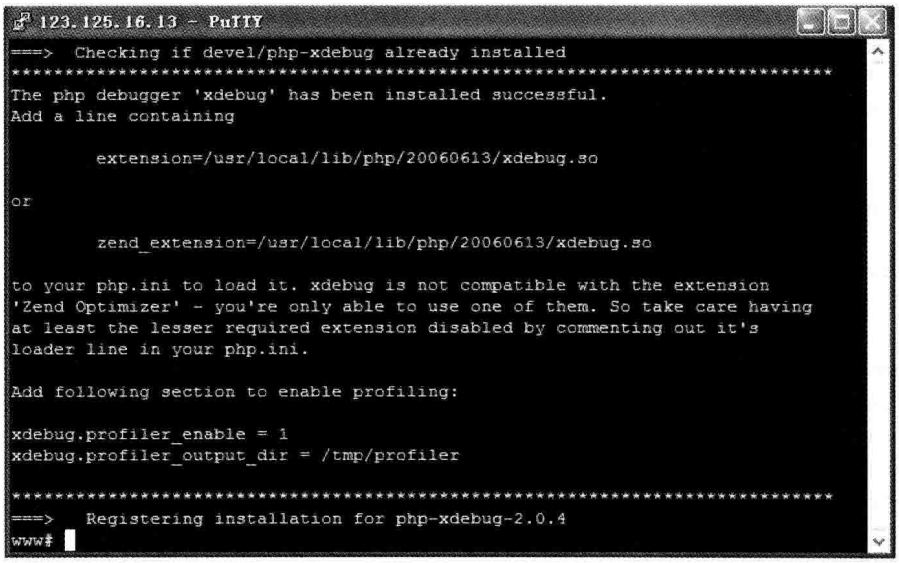


图 6-8 安装完成后的提示信息

大家可以看到 Xdebug 已经提示在 php.ini 或附属文件中增加如下行：

```
extension=/usr/local/lib/php/20060613/Xdebug.so
```

或使用：

```
zend_extension=/usr/local/lib/php/20060613/Xdebug.so
```

因为 Xdebug 是 Zend 扩展库，我们采用第 2 种写法，在 /usr/local/etc/php/extension.

ini 的其他扩展库的代码后加入，如图 6-9 所示。

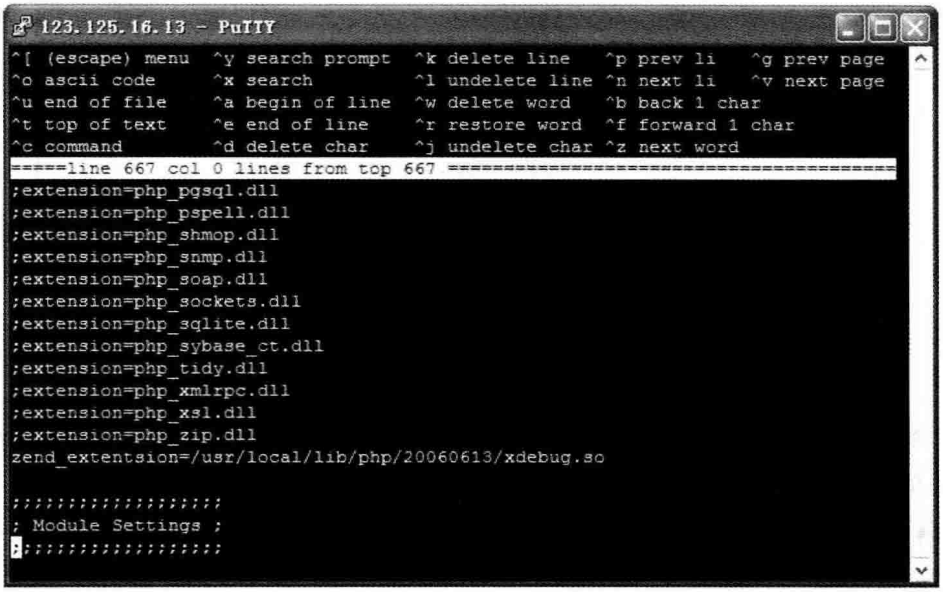


图 6-9　在其他扩展库的代码后加入

再编辑 php.ini，在尾部加入如下两行：

```
[xdebug]
xdebug.profiler_enable = 1
xdebug.profiler_output_dir = /tmp/profiler
```

接着要创建 Xdebug 输出的临时文件目录以及可写属性：

```
#mkdir /tmp/profiler
#chown www /tmp/profiler
#chmod -R 777 /tmp/profiler
```

然后使用 apachectl stop/start 命令重新启动 Apache 服务器，在 phpinfo 中就会看到 Xdebug 的信息。

3. 在 Windows 平台上安装 Xdebug

我们接下来在比较熟悉的 Windows 环境下安装 Xdebug。先去官方网站（www.Xdebug.org）下载合适的 Xdebug 版本，如图 6-10 所示，应该在前两个链接中选择一个下载，这要取决于你的 PHP5.2 是非线程安全（Non-thread-safe）版还是线程安全版，我们选择后者，因此选择图 6-10 标注的链接下载。

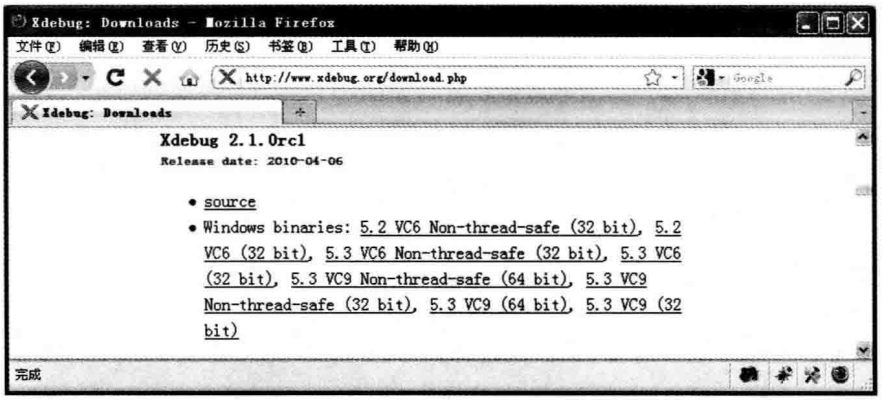


图 6-10 选择链接下载合适版本

下载后是一个名为 php_xdebug-2.1.0RC1-5.3-vc6.dll 的文件，我们可修改为更友好的文件名如 php_Xdebug.dll。

接下来开始配置，复制 php_xdebug.dll 到 $php_installed_dir/ext 目录下，修改 php.ini，在其他扩展库下增加：

```
zend_extension = "c:\php\ext\php_xdebug.dll"
```

然后在文件尾部加入如下代码段：

```
[xdebug]
xdebug.profiler_enable=on
xdebug.trace_output_dir="d:\temp\Xdebug"
xdebug.profiler_output_dir="d:\temp\Xdebug"
```

另外需要将 output_dir 事先建好，否则不会生成 profiler 数据到该目录。然后我们重新启动 Apache，看到 Xdebug 的信息，表示安装成功。

6.5.2 应用 Xdebug

我们可以自定义一个 PHP 脚本，用来专门设置 Xdebug。这样可以用该配置分别对应开发环境和生产环境。生产环境不需要太多扩展库，从而提高访问效率。示例代码如下：

```
<?php
// configure Xdebug locally
ini_set('Xdebug.var_display_max_children', 3);
ini_set('Xdebug.var_display_max_data', 6);
ini_set('Xdebug.var_display_max_depth', 2);
```

```
class DebugExample { le { ?>
```

6.5.3　Xdebug 带来的增益

当 Xdebug 已经配置正确时，我们便可以使用它提供的增强功能，都有什么呢，我们一一来看。

1. var_dump() 函数增强

首先，Xdebug 会覆盖我们的老朋友 var_dump() 函数。它改善了 var_dump() 函数的显示格式，使输出更友好，如语法突出显示等，使信息更容易看懂。

图 6-11 所示为使用 Xdebug 扩展后 var_dump() 函数与未使用 Xdebug 的 var_dump() 函数输出后的区别。

var_dump(object) & var_dump(array) Comparison

Without Xdebug

```
object(DebugExample)#1 (2) {
  ["imHiding:private"]=>
  bool(false)
  ["slf"]=>
  object(DebugExample)#1 (2) {
    ["imHiding:private"]=>
    bool(false)
    ["slf"]=>
    *RECURSION*
  }
}
array(4) {
  ["just"]=>
  string(9) "something"
  ["to"]=>
  string(6) "output"
  ["for"]=>
  string(3) "the"
  ["Debug"]=>
  string(9) "Exception"
}
```

With Xdebug

```
object(DebugExample)[1]
  private 'imHiding' => boolean false
  public 'slf' =>
    &object(DebugExample)[1]

array
  'just' => string 'someth'... (length=9)
  'to' => string 'output' (length=6)
  'for' => string 'the' (length=3)
  more elements...
```

图 6-11　使用 Xdebug 和未使用 Xdebug 的 var_dump() 函数输出的区别

下面的 Xdebug 配置设置影响 var_dump 的行为，设置 Xdebug 的功能：

xdebug.var_display_max_children、Xdebug.var_display_max_data 和 Xdebug.var_display_max_depth 是相关的 3 个设置，分别用来控制因 Xdebug.show_local_vars 的使用而显示的变量的属性数、字符串长度和嵌套深度。见表 6-2。

表 6-2　3 个设置的区别

设　置	说　明	推 荐 值
Xdebug.var_display_ max_children	显示数组和对象属性的数量	128 (128)
Xdebug.var_display_ max_data	对象属性值字符串的长度	1024 (512)

（续）

设　置	说　明	推 荐 值
Xdebug.var_display_ max_depth	显示对象或数组的嵌套深度	5 (3)

2. 错误和异常的美化

Xdebug 带来的第二个价值是，当应用程序出现未捕获的错误和例外情况时将被格式化的 HTML 显示。例如输出结果时，抛出一个异常，如图 6-12 所示。

Exception: exceptions are no good! in /Users/dirk/Sites/phpdoc/c6/debug.php on line 22

Call Stack

#	Time	Memory	Function	Location
1	0.0545	56248	{main}( )	../debug.php:0
2	0.0760	134128	DebugExample->trySomething( )	../debug.php:33
3	0.0760	134264	DebugExample->somethingWrong($bang = 'dead')	../debug.php:12

Dump $_GET

```
$_GET['var1'] = string 'test' (length=4)
```

Dump $_POST

Dump $_COOKIE

Dump $_FILES

Dump $_SESSION

Dump $_REQUEST

```
$_REQUEST['var1'] = string 'test' (length=4)
```

Variables in local scope (#3)

```
$bang = string 'dead' (length=4)
$ie = array
  'just' => string 'something' (length=9)
  'to' => string 'output' (length=6)
  'for' => string 'the' (length=3)
  'Debug' => string 'Exception' (length=9)
```

图 6-12　输出结果时抛出异常

另外，解决性能问题也是我们最需要解决的事情，这样也可获得整体性能的全面提升。

通过工具跟踪分析并修改代码，我们会积累一些经验，如文件操作、数据库交互（SQL 慢查询）、网络操作（与其他服务器之间交互的费时）、与本地服务器的交互（文件操作、数据库、邮件发送等）。另外还有如我们的函数方法是否冗余，在跟踪数据里函数被调用的次数等。

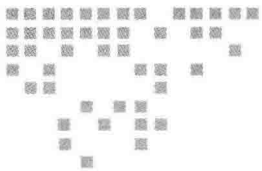
例如一个 SQL 语句花费了 10 s 才执行完毕，而我们希望将它的执行速度提升 50%，这样其实是为我们自己节省了 5 s。但是如果我们想尽办法只提高了半秒钟，则可能是受硬件、网络等限制，而不再是 CPU、内存的限制，那么我们需要筹划多台服务器的分布式部署。

另外，还有一些工具如 Xprof 等，可以做性能监测和优化建议，这里不再赘述，感兴趣的读者可以安装使用。

6.6　本章小结

在本章中，我们介绍了如何在 PHP 中使用内部函数做调试，尤其是在 PHP5 后出现的 debug_backtrace() 和 debug_print_backtrace() 等函数，这些函数只有常用才能享受到它们给我们带来的方便和价值。

在编程开发中，有一半左右的时间是在调试，会调试的程序员在开发时的效率比闷在那想的程序员效率要高很多，因为他会利用计算机来为自己做事，让代码告诉自己哪里出了问题，这样才是优秀的工程师。



Chapter 7 第 7 章

用户验证策略

无论是网站还是 APP 应用，只要有互动功能都要有会员或用户验证相关的模块，如用户注册、登录、验证，以及不同类型的用户，根据不同角色来分配权限。

如果使用 Apache、Nginx 等 Web 服务器内置的身份验证的功能有限。在更多时候，身份验证需要我们重新设计开发。

作为开发者，要充分了解 PHP 安全的必要性，特别是带有用户信息等敏感数据时，这是相当重要的资料，如果开发人员不小心，很容易留下安全隐患，如被盗取，甚至受到攻击。

在本章中，我将向各位讲述开发和存储用户账户信息的最佳做法。其中包括用户验证，掌握在网站中如何保持用户会话，并完整讲解一个访问控制列表（ACL）的技术架构。

任何良好的验证系统都需要依赖数据库。在本章的例子中，将建立一个数据库表，用来存储用户信息，然后我们在它的基础上，分别开发三个不同类型的身份验证系统。

7.1 数据库设计

用户登录验证是网站应用最常用的子系统之一。我们知道，HTTP 协议本身不维护状态，用户访问每个页面时都要进行会话维持，因此需要依赖数据库来保证会话的可靠性，

尽管如此，也需要开发人员对其进行充分优化。

在本节，我们首先定义一个数据表，稍后我会对每个字段进行解释。代码清单 7-1 为创建该表的 SQL 语句。

代码清单7-1　uses.sql

```
CREATE TABLE `users` (
`user_id` INT AUTO_INCREMENT PRIMARY KEY,
`user_name` VARCHAR(32) UNIQUE,
`email` VARCHAR(128) UNIQUE,
`password` VARCHAR(32),
`salt` VARCHAR(32),
`create_date` DATETIME,
`last_login` DATETIME
);
```

其中：该表中的 user_id 是用来存储用户的唯一主键（Primary Key）。

虽然 user_name（保存用户名）也是一个唯一值，但最好还是使用自动递增字段来作为主键。

其中这样做的主要原因就是降低用户信息和程序的耦合度。我们使用自动递增属性，这样在系统后台，管理员可以更改其用户名称，而不必担心一个用户名与其他表保持数据的参照完整性。

另外一个原因，使用数字主键也可以缩小存储空间，系统处理速度也会有所提高。

user_name 字段是网站应用中的用户名，该长度由开发者自定义，但用户名须是唯一的，我们看到已经被标识为 unique。另外把该唯一的键值添加到索引中，从而提高数据库查询时的速度。同样该道理适用于下一个称为 email 的字段。

顾名思义，email 字段是用来保存用户的电子邮件地址，在此，我们也定义邮件是独一无二的，它可以解决以下问题：

- 防止轻量级的 spam 攻击。攻击者可以编写相当简易的程序或随意填写邮件地址注册，从而攻击网站。
- 用户可能会存在很多不同的账号。唯一的电子邮件功能会提示用户已经存在该邮件，问他是否忘记了用户名或密码，可以通过电子邮件很快地找回密码。

下一个也是很重要的字段 password，用来保存用户密码。也有人会直接使用明文保存，但在不需要保存原始密码的情况下，应加密存储。常用的方法是使用 MD5 将密码单向加密为哈希数组后，再保存到该字段。

除了 MD5 外，也可以使用其他加密算法，如 SHA1 和 DES/3DES，其中 SHA1 与 MD5 一样属于单向加密，DES 为双向加密，适合于加密程度较高的场合，如银行支付以及军队等，3DES（Triple DES）是 DES 加密算法的一种模式，它使用 3 条 56 位的密钥对数据进行 3 次加密，几乎无法破解。

解密 MD5 最常见的方法是使用所谓的彩虹表（Rainbow table），听起来名字很动听，这是一个人们常用设置 MD5 后的表格，也就是一张预先计算好的明文和散列值的对照表，如果对比相同，则可能碰撞成功。

最糟糕的情况下，当用户数据库密码被破解，攻击者取得了用户注册账户，可以发现有一些用户账号密码相同，这是因为很多人在多个网站都用相同的密码，但因为数据库存储的是 MD5 哈希串，想还原这个原始密码仍然不容易。

因为一些用户使用的是相同的密码，数据库中的这些哈希字符串的值是相同的，这样攻击者利用彩虹表对照出 MD5 的明文密码，然后正常登录，取得用户资料。而 PHP 脚本则认为这是一个合法的登录用户，攻击者使用该用户数据进行购买、数据导出等行为。

RainbowCrack 是一个使用内存时间交换技术（Time-Memory Trade-Off Technique）加速口令破解过程的口令破解器，这个工具可以在地址 http://project-rainbowcrack.com/ 下载。RainbowCrack 就使用了彩虹表。

作为开发者，预防的解决方案是将插入数据库中密码字段的尾部加入一个附加字符串，这个字符串称为 salt，salt 的中文是盐的意思，试想做菜时加少许盐，菜才能有味道，密码加点盐会让它更强壮。

这个 salt 字符串可以是自定义的一个固定值，请看如代码清单 7-2 所示代码：

代码清单7-2　使用固定的salt值

```
<?php
$salt = "7dA+U^@'aF7FLvJ";
$password = 'secret';
$hash = md5($salt . $password);
echo $hash;
    /* 返回值: ade1373f24d328d9229d57f284871cd0 */
?>
```

代码中的第三行，是 $salt 字符串加上 $password 字符串后生成新的密码散列。

我们也可让 salt 字符串随机生成，这样我们自己也不知道 salt 的生成规律，安全系

数更好。如代码清单 7-3 所示：

代码清单7-3　使用随机的salt值

```
<?php
define('SALT_LENGTH', 2);    //定义salt的长度
function generate_salt($plainText, $salt = NULL) {
    if ($salt === NULL) {
        $salt = substr(md5(uniqid(rand(), true)), 0, SALT_LENGTH);
    } else {
        $salt = substr($salt, 0, SALT_LENGTH);
    }
    return substr($salt . sha1($salt . $plainText) , 0, 32);
}
$password = "test";
$salt =  generate_salt('test');
echo "salt=".$salt."<br />";
$password = md5($password + $salt);
echo 'password='. $password;
?>
```

这样我们在用户提交密码时使用该函数，插入数据库中 passwod 字段保存的是该用户密码加 salt 的 MD5 哈希值，即使攻击者有 salt 和 MD5 后的哈希，也是非常难以破解的。

另外，数据库中的字段 salt，用于用户登录时使用 salt+ 用户密码进行验证。

我们可以在整个应用程序里使用相同的 salt 值，也可以在不同的应用里使用不同的 salt 值，这样将使应用程序更不易受到攻击。

另外，用户表中为了能体现更多的状态，将来再做分析将变得有用。例如增加一个 flag 标志，用来标识是否是新账户。这样可以用来分析以下数据：

- 有多少用户注册，但从未第二次访问过网站。
- 在 1 个月内网站的活跃用户数量。
- 有多少用户登录时间超过了 1 个月。
- 有多少用户是超过一年以上的老会员。

根据应用程序，在表中还可以加入其他存储信息，例如，什么时间用户登录、真实姓名、生日、地址、邮件是否已验证等字段。

7.2　HTTP 验证

HTTP 协议提供了两种验证方法：基本验证（Base Authentication）和摘要验证（Digest

Authentication）验证。这两种方法在运行时有一些共同性和特征。

这两种方法运行时，浏览器都会弹出一个对话框，提示输入用户名和密码，作为两个参数用于验证领域，是浏览器 / 客户端在主机上登录的方法之一。

这两种验证方法与基于 HTML 网页的形式不同在于，在许多 PHP 应用程序中，在该用户脱离正常流动适用范围时需要进行身份验证。

Apache、Microsoft IIS 以及 Nginx 等主流 Web 服务器都支持这两种身份验证方法，本身并不需要 PHP。但是，使用本地 Web 服务器的方法有以下缺点：

- 注销和会话管理留给了客户端浏览器，所以是不可靠的，在时间上无法控制客户端。
- 需要额外的服务器扩展或模块，还需要结合数据库如 MySQL。
- 应用级功能将丢失，如访问控制列表（下文讨论）。

两种方法都使用 HTTP 401 来响应提示用户的登录信息。代码 401 是“未经授权”的回应。

此外，HTTP 基本验证还有一个 WWW-Authenticate 头信息被发送，它告诉客户端正在使用哪种验证方式。

7.2.1 用户名主机名验证

基本身份验证源自 HTTP 的早期，该标准定义了指定为 URL 的一部分的登录凭证，以及 HTTP 头部分的定义方法。

用户名和密码通过 URL 地址栏发送，也可以通过一个链接或书签发送，通过 URL 传递的信息，类似于如下格式：

```
http://username:password@21cto.com/
```

可以看到，有点类似于电子邮件的地址。需要注意的是，这种做法在 IE 浏览器上是不安全的，在其他浏览器中有的已经禁用这种方式。主要基于以下几个原因：

- 如果网站要求验证，客户端发送明文的用户名密码，那么网络上的窃听者可以轻而易举地获得用户名密码，起不到安全作用。
- 用户名和密码还可以被 Web 服务器记录。即使你信任服务器管理员，或你管理自己的日志，但是用户名和密码是以纯文本的形式放在 LOG 日志中的。
- 该网址还显示在客户端浏览器的历史记录。任何人都可以轻松地返回到该地址再次进行身份验证。如果在公用计算机上，这是非常不安全的。

因此这种用 URL 方式的基本验证方法，不对实体正文加密，也不能为传送的正文加上数字签名。要得到更多的安全保障，必须使用更安全的通信协议，例如 SSL（Secure Socket Layer）。

HTTP 提供了身份验证的机制，这种机制可以使通信双方互相确认身份，并且可以对传送的实体正文进行完整性校验。但是，HTTP 的验证机制仅是验证身份而已，不对实体正文加密，也不能为传送的正文加上数字签名。要得到更多的安全保障，必须使用更安全的通信协议，例如 SSL（Secure Socket Layer）。

尽管如此，HTTP 的身份验证机制也仍然是比较复杂的，RFC2616 本身没有详细解释这个机制，而是由 RFC2617 详细地描述了这个机制。

7.2.2　HTTP 的身份验证机制

很多应用都要求身份验证，例如电子商务交易，用户在网上订货，通过网上银行转账付款，这样的操作必须验证身份，不仅服务器要验证客户的身份，客户也要验证服务器的身份。

HTTP 提供了身份验证的机制，称为“质询—应答”（challenge-response）机制。当客户的请求消息要求访问被保护的资源时，服务器以状态码 401（Unauthorized）应答，并且在应答消息中包含头段 WWW-Authenticate，这个头段的值是对客户的质询。客户收到这个包含质询的消息后，再次发出请求，并且在请求消息中包含头段 Authorization，这个头段的值包含用户名和密码。服务器收到包含用户名和密码的请求消息后，加以验证，若身份合法则传送所请求的资源，否则可再次用 401 应答。

HTTP/1.1 中定义了基本验证和摘要验证这两种质询—应答方法。基本验证较容易实现，但安全性很差，用户名和密码几乎就是在网上明文传送。这种方法可用在只对资源实行简单保护、不愿付出过高代价提高安全性的场合。

摘要验证是比较安全的方法，客户和服务器用约定的加密算法对用户名和密码加密，对所传送的消息正文添加校验码以保护数据的完整性，客户和服务器还可以彼此验证身份。

在具体介绍基本验证和摘要验证之前，先明确一个概念：保护领域（realm）。

服务器上需要验证身份才能访问的资源可能是一些文件，也可能是一些在服务器上运行的程序，或服务器数据库中的数据，对于多部主机联网的服务器，也可能是其中的一部主机。这些被保护的资源可划分为若干个保护领域，供不同的用户群使用。例如，

对服务器上某个子目录设置安全权限，指定为必须验证身份才能访问，并且指定可访问这个子目录的用户群，则这个子目录就是一个保护领域。一个服务器可以设置多个保护领域，供不同的用户群使用。

一个用户可能拥有服务器上多个保护领域的访问权限，当客户要求访问被保护的资源时，服务器要求客户证实身份时必须指出资源所属的保护领域，让客户端的用户知道应该提供怎样的用户名和密码。在 HTTP 的验证机制中，保护领域用一串字符标识，这串字符是保护领域的名称，由服务器管理人员负责命名。客户在与服务器通信时，可询问终端用户，由用户根据保护领域的名称提供用户名和密码。

当客户请求访问受保护的资源时，客户请求消息 URI 指出希望访问的资源，服务器随后要求客户提供身份证明并且指出该资源所属的保护领域名称。HTTP 命名方案的 URI 是有层次结构的，按照这个结构，客户请求消息 URI 下属层次中的资源也被认为是属于同一个保护领域。

7.2.3 HTTP 基本验证

当客户向服务器发出请求消息，所请求资源属于服务器的某个保护领域，而请求消息中没有包含身份证明的头段 Authorization，或者虽然包含了 Authorization，但所提供的身份不合法时，服务器用状态码 401（Unauthorized）应答，在应答消息中包含头段 WWW-Authenticate，以此作为质询。对于基本验证方法，其语法为：

WWW-Authenticate: Basic realm=*quoted-string*

其中的 Basic 表示采用基本验证方法，*quoted-string* 表示用双引号括起来的保护领域名称，例如：

WWW-Authenticate: Basic realm="WallyWorld"

客户随后可再次发出请求，请求消息中应包含头段 Authorization，其语法为：

Authorization: Basic *Base64(username:password)*

其中的 **Basic** 表示采用基本验证方法，而 *Base64(username:password)* 的含义是：先将用户名和密码用一个冒号连接起来构成一个字符串，然后用 Base64 编码，将编码后的字符串作为头段的值。不同于 MIME，此处没有 76 个字符的限制，编码后的字符串可以是任意的长度。Base64 是 ASCII 字符的二进制数据的表示方法，它类似于 16 进制（Base16）。

例如，假如用户名是 Aladdin，密码是 open sesame，则首先连接成字符串：

Aladdin:open sesame

然后对这个字符串用 Base64 编码，编码后的数据作为字段的值：

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

在这种情况下，它主要用来作为数据混淆，虽然也很容易被解码，但这种方式要比前面用户名加密码的 URL 传递要好得多。

在本节，我们将创建一个基本验证的类，每一个验证作为一个成员方法来实现。下面的成员方法就是强制客户端进行基本身份验证。如代码清单 7-4 所示：

代码清单7-4　使用HTTP的基本验证方法

```
<?php
class baseAuthentication{
   public $username = null;
   public $password = null;
   public $unauthorizedNotice = '验证被取消';
      public function forceAuthentication(){
     header('WWW-Authenticate: Basic realm="PHP Protected Area"');
     header('HTTP/1.0 401 Unauthorized');
     die( $this->unauthorizedNotice );
   }
      public function requireAuthentication(){
           if (isset($_SERVER['PHP_AUTH_USER']) && isset($_SERVER['PHP_AUTH_PW'])) {
                  $username = $_SERVER['PHP_AUTH_USER'];
                  $password = $_SERVER['PHP_AUTH_PW'];
                  //针对于IIS服务器
           } elseif (isset($_SERVER['HTTP_AUTHENTICATION'])) {
                  if (strpos(strtolower($_SERVER['HTTP_AUTHENTICATION']),
                      'basic')===0){
                         list($username,$password) = explode(':',base64_decode
                             (substr($_SERVER['HTTP_AUTHORIZATION'], 6)));
                  }
           }
           if (is_null($username)) {
                 $this->forceAuthentication();
           } else {
                  echo "<p>你的用户名 {$username}.</p>";
                  echo "<p>你的密码 {$password}.</p>";
           }
   }
}

$auth = new baseAuthentication();
$auth -> requireAuthentication();
?>
```

该方法会发送基本身份验证头信息，浏览器这时会弹出一个友好窗口，IE 和 Firefox 会稍有不同，当用户在对话框中点击取消，会结束脚本的执行。IE 允许用户尝试验证 3 次，而 Firefox 允许用户尝试无数次，直到用户点击取消为止。

用户输入用户名和密码后，会分别保存在 $_SERVER['PHP_AUTH_USER'] 和 $_SERVER['PHP_AUTH_PW'] 数组键名中，用户点击确定后会发送给 PHP，由 PHP 处理用户输入是否正确，提交后浏览器即认为已经验证完毕，在当前会话内不会再做验证。如图 7-1 所示。

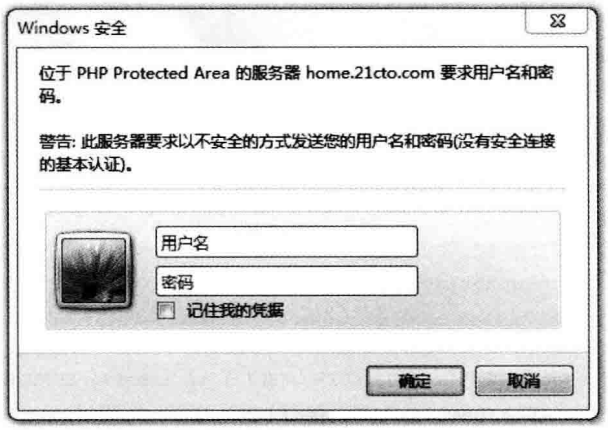


图 7-1　使用 PHP 做 HTTP 基本验证

这里，forceAuthentication 是做强制基本验证的方法，由 requireAuthentication 方法负责接受用户名和密码，如果没有输入用户名和密码，则继续要求用户填写。

另外，对于 IIS 服务器在做基本验证时，会产生 $_SERVER['HTTP_AUTHENTICATION'] 的键，将用户名密码以及验证方式加冒号以 Base64 的方式保存其中，因此需要做解码还原，如下。

```
if (strpos(strtolower($_SERVER['HTTP_AUTHENTICATION']),'basic')===0){
  list($username,$password) = explode(':',base64_decode(substr($_SERVER['HTTP_
      AUTHORIZATION'], 6)));
}
```

下面我们结合 MySQL 以及 salt 功能进行账户验证，如代码清单 7-5 所示：

代码清单7-5　结合MySQL进行基本验证

```
<?php
class baseAuthentication{
```

```
public $username = null;
public $password = null;
public $unauthorizedNotice = '验证被取消';

static public $db_host = 'localhost'; //数据库主机名
static public $db_user = 'root';        //数据库用户名
static public $db_password = '';  //数据库密码
static public $database = 'php_adv';  //数据库名称

public function requireAuthentication() {
        if ( isset($_SERVER['PHP_AUTH_USER']) && !is_null($_SERVER['PHP_AUTH_
            USER']) && isset($_SERVER['PHP_AUTH_PW']) && !is_null($_SERVER['PHP_
            AUTH_PW'])) {
                session_start();
                $user = $_SERVER['PHP_AUTH_USER'];
                $password = $_SERVER['PHP_AUTH_PW'];
                if ( $_SERVER['PHP_AUTH_USER'] != $_SESSION['user'] || $password
                    != $_SESSION['password'] ) {
                                $conn = mysql_connect(self::$db_host,self::
                                    $db_user,self::$db_password);
                                mysql_select_db(self::$database, $conn );
                                $query = "SELECT * FROM `users` WHERE `user_
                                    name`='%s'";
                                $query = sprintf($query, mysql_escape_string
                                    ($_SERVER['PHP_AUTH_USER']));
                                $userinfo = mysql_fetch_assoc( mysql_query
                                    ($query) );

                                //如果存在该用户
                                if($user==$userinfo['user_name']){
                                        $saltedPassword = md5($password +
                                            $userinfo['salt']);
                                        if($saltedPassword == $userinfo
                                           ['password']){
                                                $_SESSION['user'] = $_SERVER
                                                    ['PHP_AUTH_USER'];
                                                $_SESSION['password'] = $salted-
                                                    Password;
                                                echo '欢迎你登录: '.$_SERVER ['PHP_
                                                    AUTH_USER'];

                                        }else{

                                                echo('错误的密码');
                                        }
```

```
                                    }else{
                                            echo('没有该用户');
                                    }
                                    mysql_close($conn);
                            }
                    } else {
                                    $this->forceAuthentication();
                    }
            }

            public function forceAuthentication(){
                    header('WWW-Authenticate: Basic realm="PHP Protected Area"');
                    header('HTTP/1.0 401 Unauthorized');
                    die( $this->unauthorizedNotice );
            }
    }

    $auth = new baseAuthentication();
    $auth -> requireAuthentication();
    ?>
```

一个完整的应用程序应该还可以跟踪无效密码次数，当尝试次数达到限制时，及时通知用户。

7.2.4 摘要访问验证

与基本验证方法相比，摘要验证方法相对要复杂一些。

首先，服务器和客户必须选择一种算法，这种算法可以将任意长度的数据映射为一个摘要，并且这种算法必须是不可逆的，即不能根据摘要计算出原来的数据。满足这种要求的算法有很多，有很多散列函数（hash）都可满足这个要求。

首先，当客户发出请求时，服务器仍然发出质询，例如：

HTTP/1.1 401 Unauthorized

WWW-Authenticate: Digest

realm="test@foo.com",

algorithm=<*algorithm-name*>

其中的 Digest 指出，服务器要求用摘要验证方法，realm="test@foo.com" 指出保护领域的名称，algorithm 指出计算摘要的算法，realm、algorithm 统称为指令（directive），摘要验证需要用很多指令，以后将逐一介绍。

HTTP/1.1 规定，如果没有用指令 algorithm 指定算法，则默认为使用 MD5。MD5 是一种散列算法，它将任意长度的数据映射为 128 位二进制数的摘要，算法不可逆，并且任何输入数据的微小变化就可以导致完全不同的摘要，这使得恶意的攻击者不可能根据摘要猜测原来的数据。

与基本验证方法一样，客户再次发送请求消息，其中包含头段 Authorization。设指令 algorithm 指定的算法为函数 H，客户可将用户名、密码和保护领域名称连接成一个字符串，彼此用冒号隔开，利用 H 计算摘要：

digest = H(*username:password:realm*)

然后将 digest 作为 Authorization 的值传送给服务器。但是 digest 一般是二进制数，必须转换为字符串才能包含在头段中。转换的方法是将每 4 位二进制数用一个对应的 16 进制数字字符表示，其中字母用小写。例如，10011100 转换为“9c”。MD5 的摘要是 128 位二进制数，因此将转换为 32 个 16 进制数字。以下我们用到函数 H 时，都是认为函数值已按这种方法转换为字符串。

Authorization 的格式现在要复杂一些，例如：

Authorization: Digest

username=<*user-name*>,

realm=<*realm-name*>,

response=*request-digest*

其中指令 username 的值 <*user-name*> 是实际的用户名，realm 的值 <*realm-name*> 是保护领域名称，response 的值 *request-digest* 是转换为字符的摘要。注意用户名是公开的，但是密码是不公开的。

服务器接收到消息后，当然服务器无法根据消息中的摘要计算密码，但是服务器是知道用户密码的，只需要重新计算 H(*username:password:realm*)，然后与客户送来的摘要比较，两者一致就可以认定客户是合法的。

由于 H 不可逆，所以其他人不可能根据 digest 获知用户名和密码。但是，仅仅这样是不行的，恶意攻击者并不需要知道密码，他们可以很容易地将摘要复制下来，然后冒充合法用户向服务器发出请求，由于摘要是按合法用户的密码计算出来的，所以服务器将会上当受骗。这种欺骗方式称为“重播”(replay)。

解决的方法是由服务器在发出质询时同时加上一个“现时”数据（nonce），这个数据必须是随时变化的，客户和服务器之间的每次通信都使用不同的 nonce。RFC2617 建议，

用如下方式构造数据 nonce：

nonce=Base64(*time-stamp* H(*time-stamp*:*ETag*: *private-key*))

其中 Base64(…) 表示将括号中的二进制数据用 Base64 算法转换为字符串，*time-stamp* 表示执行运算时的时间戳，H 是计算摘要的散列算法（例如 MD5），*ETag* 是将要传送的实体的标签，*private-key* 是服务器才知道的密钥。

服务器在发出质询时用指令 nonce 指出现时数据是什么，例如：

WWW-Authenticate: Digest

realm="test@foo.com",

algorithm="MD5",

nonce="hsggh87jfdii2m4n4mi34h4j45n"

客户收到质询消息后，用用户名、密码连同 nonce 构造摘要。为描述算法，我们使用 RFC2617 中使用的符号：KD(a, b)、A1 和 A2。

KD(a, b) = H(a:b)

即 KD(a, b) 表示将两个字符串 a、b 用冒号连接成一串，然后用散列算法 H 计算的摘要。而 A1、A2 则分别为

A1 = *username:realm:password*

A2 = *Method:digest-uri*

（注：A1、A2 各有两种不同的算法，此处是其中的一种。）

即 A1 是用户名、保护领域名称和密码用冒号连接起来的一串字符，A2 是客户请求消息中的请求操作名称（GET、POST 等）和请求消息中的 URI 用冒号连接起来的一串字符。

于是客户证实身份的摘要可按下式计算：

request-digest = "KD(H(A1), nonce:H(A2))"

现在客户可以用 Authorization 向服务器发送证实身份的消息：

Authorization: Digest

username=<*user-name*>,

realm=<*realm-name*>,

nonce=<*nonce-value*>,

uri=<*digest-uri*>,

response=*request-digest*

其中的 *nonce-value* 是服务器送来的现时数据，*digest-uri* 是客户请求消息请求行中

的 URI，之所以要加上指令 uri 重复指出这个 URI，是因为客户和服务器之间的代理服务器或网关有可能更改请求消息请求行中的 URI。

注意除了密码之外，现在包含 Authorization 的请求消息中包含所有重新计算 *request-digest* 所需要的信息，服务器收到消息后，只需按照其中提供的信息以及已知的用户密码，就可以方便地重新计算 *request-digest*，如果与客户送来的相同，则证明客户是合法的。而对于使用重播方法的攻击者，虽然可以看到 nonce，但是不知道密码，无法计算出 *request-digest*，又由于服务器每次都使用不同的 nonce，所以也不能复制以前的 *request-digest*。

最危险的攻击来自客户和服务器之间的恶意代理服务器，这种攻击者称为“中间人”（man in the middle）。中间人可以在客户和服务器的一次通信中，拦截客户证实身份的消息，复制其中的 *request-digest*，然后冒充客户向服务器要求访问其他受保护的资源。nonce 中包含的实体标签 ETag 和 A2 中包含的 URI 可以有效地防止这种攻击。

但是问题并没有到此结束。中间人可以使用所谓“明文选择攻击法”（chosen plaintext attack）获知用户密码。首先将常用的、猜测的密码编成一本字典，其中可能包含数百万的条目，攻击者先选定一个 nonce，按照这个 nonce 计算字典中每一条目对应的摘要，然后冒充服务器向客户发出质询，其中包含事先选定的 nonce，收到客户的应答之后，比较客户的摘要和事先计算好的摘要，如果某一条目的摘要与客户的摘要相同，则这一个条目就是客户的密码。对付明文选择攻击没有彻底的解决办法，只能要求用户使用较长且不易猜测的密码。另外，客户也可以提供自己的现时数据 cnonce，在计算摘要时将 cnonce 包含进去，当然，为了使服务器能够重新计算摘要，必须在请求消息中指出 cnonce 是什么。由于攻击者事先不知道 cnonce，也就无法事先计算摘要，虽然可以拦截客户消息后根据已知的 cnonce 重新计算摘要，但这样的计算量可能使得攻击者无法承担。

客户的身份经证实后，可能会继续访问保护领域内的其他资源，但是 HTTP 是没有状态的（即 HTTP 每次请求—应答都是彼此独立的），下次访问服务器并不知道客户已在上次证实过，于是又要重新发送 nonce，客户又必须重新计算摘要。为提高效率，可以考虑让客户重复使用一个 nonce，服务器在开始发出质询时再加上一个会话标识字符串 opaque（由服务器自己设定），客户在随后的请求中重复使用先前 nonce 计算摘要，并且附上 opaque 说明是同一个会话过程的延续。这样客户不必在每次发出请求消息之前都等待服务器发来新的 nonce，减少了往返来回的通信。但是这种做法会招致重播攻击，不

过，如果客户的请求操作是 GET，则不会有重播攻击的风险。首先，HTTP 的质询—应答机制只是用来证实客户身份，并没有对传送的消息正文加密，客户第一次通过身份验证后，所获得的实体正文在网络上是明文传送的，网络上任何机器都可以看到正文，因此重播获得同样的实体正文是没有意义的。而如果攻击者冒充客户企图获得其他资源，则由于摘要中已包含请求 URI，如果请求消息请求行中的 URI 与摘要中的 URI 不符，服务器将返回 400（Bad request），所以也是不能奏效的。但是，如果客户的请求操作是 POST 或者 PUT，则重播就可能产生严重的后果。为此，可在 Authorization 中增加一条指令 nc，nc 的值是 nonce 的使用计数，客户从 nc=1 开始，每重复使用一次 nonce，就把 nc 的值加 1，如果有攻击者重播，服务器就会收到 nc 重复的请求，服务器可以将这种请求作为重播攻击对待。

虽然 HTTP 的验证机制不对消息正文加密，但是可以利用这个机制保证正文的完整性。所谓完整性是指正文内容没有被修改，无论这种修改是网络的偶然错误还是恶意的攻击，都可能造成严重的后果。例如，一个使用 HTTP 与远程客户通信的数据库，如果客户提交的数据被修改，则写入数据库之后就可能导致严重的问题。HTTP 保证完整性的方法是计算 H(entity-body)，即用实体正文作为输入数据计算散列函数值，然后将计算结果包含在 A2 中。

以上仅考虑了服务器对客户的身份验证，然而客户也要验证服务器是否是假冒的，为实现客户对服务器的验证，服务器可在验证客户身份后，在发送应答消息时包含头段 Authentication-Info，其中用所知道的用户密码同样计算一个摘要，客户在收到服务器的摘要后，也用自己的密码计算摘要，如果两者一致，则证明服务器确实知道客户密码，服务器不是假冒的。

综上所述，对于摘要验证方法，HTTP/1.1 对服务器的质询头段 WWW-Authenticate 和客户的应答头段 Authorizaion，以及服务器证实自身的头段 Authentication-Info 的完整定义如下，其中有些规定是为了保持与 HTTP/1.0 兼容。

服务器质询客户的头段 WWW-Authenticate 的语法是：

WWW-Authenticate: Digest

realm="*<realm-value>*"

nonce="*<nonce-value>*"

[**domain=**"*<list-of-URIs>*"]

[**opaque=**"*<opaque-string>*"]

[stale=(true | false)]

[algorithm=(MD5 | MD5-sess | *token*)]

[qop=("auth" | "auth-int" | "auth,auth-int")]

[*<extension-directives>*]

其中各条指令的含义如下：

realm 是保护领域名称。

nonce 是现时数据，每次服务器用 401 提出质询时，必须提供不同的 nonce，建议这个数据采用 Base64 编码或者 16 进制字符。客户只需复制使用 nonce，不必理解 nonce 的构造。

domain 的值是若干个 URI，彼此用空格分隔。domain 中的 URI 都属于 realm 保护领域，客户可根据这些 URI 判断哪些资源属于 reaml 指出的保护领域。注意按照 HTTP URI 的结构，所有属于这些 URI 层次结构下的资源也属于这个保护领域。如果没有 domain 指令，客户则应假定服务器上所有资源都属于这个保护领域。

opaque 可作为客户和服务器之间的会话标识，客户应该在随后请求同一个保护领域中的资源时，在 Authorization 中包含同样的 opaque。

stale 若为 true，则说明客户先前的请求由于使用的 nonce 已过时而被拒绝，客户必须使用新的 nonce 重新计算摘要。

algorithm 指出计算摘要的算法，此处定义了“MD5”和“MD5-see”，它们的区别在于 A1 的计算方法，稍后介绍。客户和服务器也可以选择双方理解的其他算法，*token* 表示其他算法的名称。这条指令是可选的，若没有则默认为使用 MD5。

qop 是 quality of protection 的缩写，qop 的值 auth 表示仅对用户身份进行验证，auth-int 表示不仅要验证身份，还要保护实体正文的完整性。如果 auth 和 auth-int 两项都列出，则表示服务器对这两种方法都支持，由客户选择其中一种。qop 的值决定客户计算 A2 的方法，稍后介绍。

<extension-directives> 表示以后可能扩充的指令。

客户证明身份的头段 Authrization 的语法是：

Authorization Digest

username="*<username-value>***"**

realm="*<realm-value>***"**

nonce="*<nonce-value>***"**

uri=<*request-uri*>

responce="<*request-digest*>"

[**algorithm=(MD5 | MD5-sess |** *token*)]

[**cnonce="**<*cnonce-value*>"]

[**opaque="**<*opaque-value*>"]

[**qop=("auth" | "auth-int")**]

[**nc=**<*nonce-count*>]

[*auth-param*]

其中各条指令的含义如下：

username 是客户的用户名。

realm 是服务器发来的保护领域名称。

nonce 是服务器发来的现时数据。

uri 是客户请求消息请求行中的 URI，此处重复这个 URI 是因为代理服务器可能会改变请求行中的 URI。

algorithm 是服务器 WWW-Authenticate 中指定的算法。

cnonce 是客户自己的现时数据，用来避免明文选择攻击。为了保持与 HTTP/1.0 兼容，必须在使用 qop 指令时才能使用 cnonce。

opaque 是服务器送来的数据，必须原样复制。

qop 是客户选定的保护方法，必须是服务器 WWW-Authenticate 中 qop 指令指出的方法。qop 是可选的，这是为了与 HTTP/1.0 兼容。

nc 是客户重复使用 nonce 时的使用次数（从 1 数起，包括当前这一次），用 16 进制数表示，用来避免重播攻击。只有本头段同时使用 qop 时才能使用 nc，另外，如果服务器的 WWW-Authenticate 中没有使用 qop，也不能使用 nc。

[*auth-param*] 是留待以后扩充的指令。

responce 最重要，其值 request-digest 是客户证明身份的摘要，其计算公式如下：

设 algorithm 指定的算法为 H，则当没有 qop 指令时，

request-digest =KD(H(A1), *nonce-value*:H(A2))

当 qop="auth" 或 qop="auth-int" 时，

request-digest =KD(H(A1), *nonce-value*:*nonce-count*:*cnonce-value*:*qop-value*:H(A2))

其中 *qop-value* 为 auth 或 auth-int，A1 的计算公式为：

如果 algorithm 为 MD5，则

A1=*username-value*:*realm-value*:*password*

其中 *password* 是用户密码。

如果 algorithm 为 MD5-sess，则

A1=H(username-value:realm-value:password):nonce-value:cnonce-value

A2 的计算公式为：

如果没有 qop 或 qop="auth"，则

A2=*Method*:*request-uri*

其中 Method 是当前请求行中的操作方法。

如果 qop="auth-int"，则

A2=*Method*:*request-uri*:H(*entity-body*)

其中 *entity-body* 表示请求消息的实体正文，H(*entity-body*) 是实体正文的完整性校验码。服务器在接收到含有 Authorization 的消息后，将重新计算 *request-digest*，为此必须重新计算 A1 和 A2，如果实体正文被修改，则 H(*entity-body*) 将会不一样，从而 A2 也不一样，最后得到的 *request-digest* 将与客户送来的不同，于是访问将被拒绝。

服务器在证实客户的身份后，将执行客户请求的操作并发出应答消息，在应答消息中可包含头段 Authentication-Info，这个头段一方面可为客户与服务器的下一次“请求—应答”做准备，另一方面可向客户证实服务器的身份。Authentication-Info 的语法如下：

Authentication–Info: nextnonce="<*nonce-value*>"

[**qop=("auth" | "auth–int")**]

[**cnonce=**"<*cnonce-value*>"]

[**nc=**<*nonce-count*>]

[**repauth=**"<*response-digest*>"]

其中 nextnonce 是服务器为客户下次计算摘要提供的现时数据，这样客户在收到应答消息后，马上可以计算下次请求需要的摘要，不必等待服务器再次发来现时数据。这种方法与用户重复使用 nonce 的方法一样，可以提高效率，而且又可以防止重播。但是这种方法也有缺点，为了提高通信速度，有时候客户会采用“流水线”方式与服务器通信，即连续向服务器发送若干个请求消息，而不是发出一个请求消息后，等待服务器的应答消息，收到后再发下一个请求消息。使用 nextnonce 将使得客户不能采用“流水线”方式。

qop、cnonce和nc都是对与所应答的客户请求消息中相同指令的复制。

repauth是服务器的身份证明，repauth的值*response-digest*也是一个摘要，计算方法和客户的*request-digest*计算方法相同，只是其中A2的计算公式有所不同：

如果qop="auth"或没有使用qop，则

A2=:*request-uri*

如果qop="auth-int"，则

A2=:*request-uri*:H(*entity-body*)

其中*request-uri*是所应答的客户请求消息中uri指令的值，*entity-body*是服务器应答消息中所包含的实体正文。与客户端A2的计算公式比较就可以知道，只是少了Method一项而已。服务器的应答消息是没有Method的。

由于*response-digest*中包含客户密码，所以假冒的服务器是无法计算正确的摘要的。客户收到应答消息后，可重新计算*response-digest*，如果与服务器送来的一致，就说明服务器确实知道客户密码，是真实的服务器。

基本验证和摘要访问验证都是很脆弱的。基本验证可以让窃听者直接获得用户名和密码，而摘要访问验证窃听者只能获得一次请求的文档。

下面描述如何在PHP中实现摘要验证。我们编写一个叫作DigestAuthentication的验证类，如代码清单7-6所示：

代码清单7-6 DigestAuthentication类

```
<?php
class DigestAuthentication{
    private $realm = 'The batcave';

    //强制弹出认证窗口
    private function requireLogin($realm,$nonce) {
      header('WWW-Authenticate: Digest realm="' . $realm . '",qop="auth",
          nonce="' . $nonce . '",opaque="' . md5($realm) . '"');
      header('HTTP/1.0 401 Unauthorized');
      echo '你选择了取消按钮';
      die();
    }

    public function requireAuthenticate(){
            // 生成随机数
            $nonce = uniqid();

            // 从http 头中取得签名
```

```
        $digest = $this->getDigest();

        // 如果不是签名，再次认证
        if (is_null($digest)) {
          $this->requireLogin($realm,$nonce);
        }

        $digestParts = $this->digestParse($digest);
        $validUser = 'admin';
        $validPass = '1234';

        //根据我们搜集的所有信息，我们可以计算出返回的是什么
        $A1 = md5("{$validUser}:{$realm}:{$validPass}");
        $A2 = md5("{$_SERVER['REQUEST_METHOD']}:{$digestParts['uri']}");

        $validResponse = md5("{$A1}:{$digestParts['nonce']}:{$digestParts['nc']}:
            {$digestParts['cnonce']}:{$digestParts['qop']}:{$A2}");
        if ($digestParts['response']!=$validResponse){
            $this->requireLogin($realm,$nonce);
        }

        // 登录成功
        echo '您好，恭喜登录成功!';
}

// 该方法返回一个签名字符串
private function getDigest() {
  // 当PHP以Apache模块化安装时
  if (isset($_SERVER['PHP_AUTH_DIGEST'])) {
       $digest = $_SERVER['PHP_AUTH_DIGEST'];
       // 如果不是，代表是其他Web服务器
  } elseif (isset($_SERVER['HTTP_AUTHENTICATION'])) {
        if (strpos(strtolower($_SERVER['HTTP_AUTHENTICATION']),'digest')===0)
            $digest = substr($_SERVER['HTTP_AUTHORIZATION'], 7);
  }
  return $digest;
}

// 提取签名信息
private function digestParse($digest) {
  // 防止数据丢失
  $needed_parts = array('nonce'=>1, 'nc'=>1, 'cnonce'=>1, 'qop'=>1, 'username'=> 1,
      'uri'=>1, 'response'=>1);
  $data = array();
  preg_match_all('@(\w+)=(?:(?:")([^"]+)"|([^\s,$]+))@', $digest, $matches,
```

```
            PREG_SET_ORDER);
        foreach ($matches as $m) {
            $data[$m[1]] = $m[2] ? $m[2] : $m[3];
            unset($needed_parts[$m[1]]);
        }
        return $needed_parts ? false : $data;
    }

}
//使用该类的实例
$d = new DigestAuthentication ();
$d->requireAuthenticate();
?>
```

以上代码，实现了验证处理，但是用户名密码是固定在代码里的，下面我们将 requireAuthenticate() 方法改写以实现与数据库中记录的验证，如代码清单 7-7 所示：

代码清单7-7 加入数据库验证方法

```
public function requireAuthentication() {
    if ( !($directives = $this->parseDigest($_SERVER['PHP_AUTH_DIGEST'])){
                        $this->forceAuthentication();
    }
    if ( $this->getValidDigest($directives) != $directives['response'] ){
                $this->forceAuthentication();
    }
}
private function getValidDigest( $directives ) {
    $conn = mysql_connect( self::$db_host,
    self::$db_user,
    self::$db_password );
    mysql_select_db( self::$database, $conn );
    $query = "SELECT * FROM `users` WHERE `username`='%s';"
    $query = sprinf($query, mysql_escape_string($directives['user']));
    $user = mysql_fetch_assoc( mysql_query($query) );
    mysql_close($conn);
    if ( !$user ) return false;
    $A1 = md5($directives['username'] . ':' .
    $this->realm . ':' . $user['password']);
    $A2 = md5($_SERVER['REQUEST_METHOD'].':'.$directives['uri']);
    $validDigest = md5($A1.':'.$directives['nonce'].':'.
    $directives['nc'].':'.$directives['cnonce'].':'.
    $directives['qop'].':'.$A2);
    return $validDigest;
}
```

因为 HTTP 验证是 Web 服务器内置的功能，使用起来很方便也很有用，使用签名验证的安全性也值得我们信任。

但是 HTTP 验证的缺点是定制化和可扩展化不够好，比如登录框的外观不能定制，返回消息也需要依赖浏览器，因此我们更多时候使用 PHP 编写验证脚本。

7.3　纯 PHP 验证

纯 PHP 验证与标准 HTTP 方法不同，它不再使用 HTTP 头信息，而是将用户名和密码的 HTML 表单嵌入页面中。

当用户输入用户名密码信息时，将使用 Session 和 Cookie 技术来维护用户会话。另外还可以使用第三方用户通行证（Passport）服务，如 Google OAuth 和 OpenID(http://www.openid.org)。

使用 PHP 验证实质是应用程序和服务器之间的握手验证，需要在安全方面采用适合的预防措施，否则恶意用户很容易劫持用户会话，如下方式：

- 通过一个跨站脚本或用户 Cookie 攻击服务器。由于 Cookie 是正确的，网站应用程序无法识别它的正确性，则认为是合法用户。
- 采用嗅探网络上的数据包，取得 Cookie 数据，按正常的方式发送给网站发送 Cookie，成为合法登录用户。

这些攻击基本上都采用重放攻击方式，本节重点将讨论如何防止用户名和密码被盗走，包括使用服务器端的 Session 来存储用户信息。然后讲述如何将用户信息加密保存在 Cookie 中。

7.3.1　自定义 Session

PHP 的 Session 会话是将用户信息存储在服务器上。一般情况下，一个 SessionID 会存储在客户端的 Cookie 中，然后通过 SessionID 的时间维护 Session 生命周期的建立和保持。

Session 处理中包括一组函数，最常见的是 session_start() 函数。Session 生命周期开始后会将初始化的数据存储在 $_SESSION 全局数组，其间的会员数据会在该数组中反映。

Session 也可不需使用 cookie，SessionID 可以通过 GET 和 POST 变量传递。该验证设置可以在 php.ini 配置文件中修改。

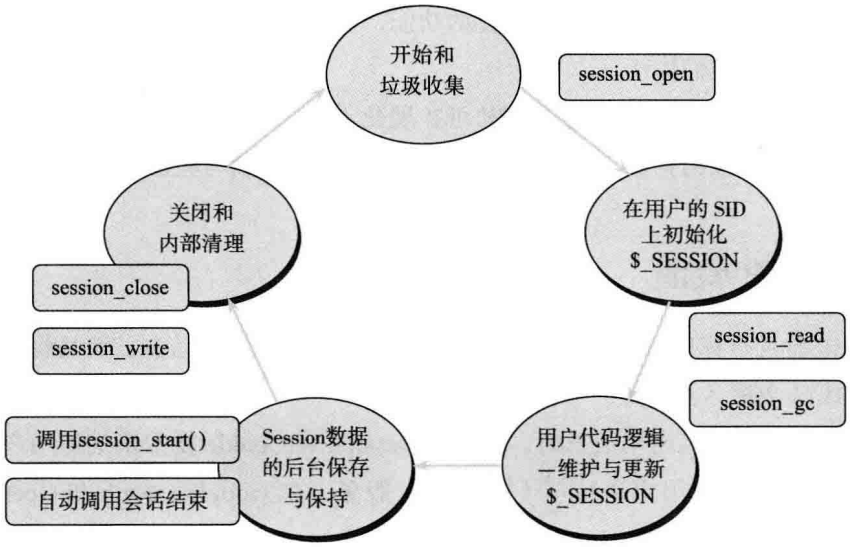


图 7-2 Session 的生命周期

默认的 Session 数据文件是存储在服务器的某个目录中，我们可以设置 Session 使用不同的存储机制，如将 Session 保存到 MySQL 中，或者使用 memcached 将 Session 保存到内存中。

下面我们使用数据库存储 Session 数据，并编写一个类。以下详细说明：

首先，MySQL 数据库的表设计结构如下：

```
CREATE TABLE `session` (
`sessionid` VARCHAR(128),
`uid` INT(13),
`data` MEDIUMBLOB,
`timestamp` INT(14),
`ip` VARCHAR(15),
PRIMARY KEY ( `sessionid` ),
KEY ( `timestamp`, `sessionid` )
)
```

最后一个字段 ip 用来保存 IP，主要功能是安全设置，防止会话攻击，如果有人企图模仿 Session，可以校验 IP 地址进行有效阻止。

timestamp 字段用于处理到期的旧 Session，后面我们详细介绍。

如果要使用自定义 Session 处理程序，必须在 Session 开始后运行。一般通用的做法是在每个脚本的前面使用 include，将 session.php 文件包含进来，这样 PHP 脚本运行时将先运行包含文件。

自定义 Session 接口函数为 session_set_save_handler()。该函数有 6 个参数，这些参数都是用于函数的回调。格式：

```
session_set_save_handler( 'custom_session_open' , 'custom_session_close' ,
                          'custom_session_read' , 'custom_session_write' ,
                          'custom_session_destroy' , 'custom_session_gc' );
```

第一个回调函数是 open()，在本例中该函数将建立一个 MySQL 数据库连接。Open 方法有两个参数。一个是保存的路径，一个是 SessionID，该 ID 用于数据表的主键。

第二个方法是 close()，关闭方法，当脚本完成 Session 读写后并不打算在使用时调用它。

值得一提的是关于 Session 锁定，当不想让脚本对 Session 进行修改，变成只读，可使用 session_write_close() 函数。这种方法我们将在后面讨论，详细内容请查看 CustomSession 类。

1. 使用 MySQL 保存 Session

read() 和 write() 函数功能是相辅相成的，分别用于读取和写入 Session 数据。在本例中，打开功能是一个 MySQL 的 SELECT 查询，写功能是 INSERT 或 UPDATE 查询。

destory 函数用于手动销毁 Session，它只删除与该 Session 相关的所有数据，不会破坏其他 SessionID 或 Cookie。

过时的会话需要定期从系统中清除，这就是专门负责垃圾收集的 gc 函数。

请注意，它会执行定期清理过时会话任务，gc 执行的时间与 php.ini 中的 3 个参数：session.gc_maxlifetime，session.gc_probability 和 session.gc_divisor。

垃圾收集器会在指定时间，删除所有已经超过生命周期的会话，PHP 默认为 24 分钟（或 1440 秒），即客户端超过 24 分钟没有刷新，当前 Session 就会失效。

注意　如果超时不删除过时的 Session 可能是一个非常耗时的行为，尤其是有几百万的访问时。我们在设计表时使用了一个复合键 ID，这样避免多个用户登录的情况下产生冲突。

我们使用数据库管理 Session 后，就可以让 Session 持久，不再受以上时间的约束，该 Session 管理类的完整代码如代码清单 7-8 所示：

代码清单7-8　完全的Session管理类

```
<?php
```

```
class CustomSession {
    private static $db_host = "localhost";
    private static $db_user = "root";
    private static $db_password = "";
    private static $database = "sessions";
    private $conn;

    public static function getInstance() {
            static $instance = null;
            if ( $instance == null ) {
                    $instance = new CustomSession();
            }
            return $instance;
    }

    public function __construct() {
            session_set_save_handler(
            array($this,"open"), array($this,"close"),
            array($this,"read"), array($this,"write"),
            array($this,"destroy"), array($this,"gc") );
    }

    public function __destruct() {
            session_write_close();
    }

    public function open( $path, $id ) {
            $this->conn = mysql_connect( CustomSession::$db_host,
            CustomSession::$db_user,
            CustomSession::$db_password );
            mysql_select_db( CustomSession::$database, $this->conn );
    }

    public function close() {
            mysql_close($this->conn);
    }

    public function read( $id ) {
            $escaped_id = mysql_escape_string( $id );
            $res = $this->query("SELECT * FROM `session` WHERE `sessionid`=
                '$escaped_id'");
            if ( $row = mysql_fetch_assoc( $res ) ) {
                    $this->query("UPDATE `session` ".
                                    "SET `timestamp` = UTC_TIMESTAMP() ".
                                    "WHERE `sessionid`='$escaped_id'");
```

```
                return $row['data'];
        }
        return "";
    }

    public function write( $id, $data ) {
        $query = "REPLACE INTO `session` ".
                    "(`sessionid`, `data`, `ip`, `timestamp`) ".
                    "VALUES ('%s', '%s', '%s', UNIX_TIMESTAMP(UTC_TIMESTAMP()))";
        $this->query(sprintf( $query, mysql_escape_string($id),mysql_
            escape_string($data),$_SERVER["REMOTE_ADDR"] ) );
    }

    public function destroy( $id ) {
        $escaped_id = mysql_escape_string( $id );
        $res = $this->query( "DELETE FROM `session` WHERE `id`='$escaped_id'" );
        return ( mysql_affected_rows($res) == 1 );
    }

    public function gc( $lifetime ) {
        $this->query( "DELETE FROM `session` WHERE ."UNIX_TIMESTAMP(UTC_
            TIMESTAMP())-`timestamp` > $lifetime");
    }

    public function query( $query ) {
        $res = mysql_query($query, $this->conn);
        return $res;
    }
}
?>
```

请留意在保存 Session 数据时，我们是保存的序列化的字符串，因为会话数据既有字符串也可能包括二进制数据，正是此原因，在数据库使用 BLOB 二进制类型的字段，这样不仅保证数据安全，而且占用的存储空间也会变少。

2. 使用 memcached 保存 Session

在之前的内容中，我们介绍过 memcached 这个分布式内存数据库。那么我们也可通过它把 Session 数据保存在内存中，同样会提升网站的访问速度。

有关安装和配置以及技术细节，请各位参考缓存管理 memcached 一章之内容。

下面是一个 memcached 会话管理类，如代码清单 7-9 所示：

代码清单7-9 使用memcached管理Session会话

```
<?php
class MemcacheSessionHandler{
    private $lifetime = 0;
    private $mc = null;

    public function start()       {
            if(!session_set_save_handler (array(&$this,'open'),
                    array(&$this,'close'),
                    array(&$this,'read'),
                    array(&$this,'write'),
                    array(&$this,'destroy'),
                    array(&$this,'gc')))
            return false;
            session_start();
    }

    public function __construct(){ }
    public function __destruct(){
            session_write_close();
    }
    public function open(){
      $this->lifetime = ini_get('session.gc_maxlifetime');
            /*self::$mc = new Memcache;
            self::$mc->connect('127.0.0.1');*/
      $this->mc = Cache::getMc(); // this is my helper class for memcache
      return true;
    }

    public function read($id)     {
        return $this->mc->get("sess_{$id}");
    }

    public function write($id, $data)     {
        return $this->mc->set("sess_{$id}", $data, $this->lifetime);
    }

    public function destroy($id){
            return $this->mc->delete("sess_{$id}");
    }

    public function gc(){ return true; }
    public function close(){ return true; }
}
```

```
/* 使用方法 */
session_name($mySessionKey);
$gc_maxlifetime = 2592000; // 30天
session_set_cookie_params($gc_maxlifetime);
ini_set('session.gc_maxlifetime', $gc_maxlifetime);

$sh = new MemcacheSessionHandler();
$sh->start();
?>
```

PHP 5.1 以后，close 和 write 处理器将会在类析构函数之后调用，因此处理过程中不能使用类和抛出异常（Exception 也是一个类）而在析构函数内可以使用 Session，因此我们需要在程序中（比如在 open 或者 read 处理器内）声明一下保证让 session_write_close 在析构方法执行前完成。

在数据库保存用户 ID 和 Session 后，使用 PHP 很容易知道当前哪个用户在线，哪些用户在网站的哪个页面上动作，这些行为在很多应用上都会应用到。

会话过期处理。会话过期将不再保存该数据，也不能够再使用，此种行为适于确定已登录用户的状态，以及他们的一些信息，如电子商务网站的购物车、订单，在线客服的聊天时间，等等。

当 Session 保存在数据库或内存后，可以设置为不受 1440 秒的限制，这样通过 Cookie 和 Session 配合，可以保存用户很长时间的动作数据，如点评过的商铺、已查看的商品、已播放过的视频等。

7.3.2　构造安全的 Cookie

在一些很多网站，当用户登录后，会将登录的相关信息以纯文本的形式放在 Cookie 中，这样很容易地被攻击者查看并复制，像一把被复制的钥匙，攻击者很容易模仿 Cookie 登录网站，访问正常的用户功能和数据。因此需要对以下 Cookie 的内容进行加密处理：

- User ID。
- 用户最后登录的 IP 地址。
- 一个时间戳。

第一项用来了解是哪些用户登录。

第二项是用来记录最后登录的 IP 地址，可以用它与服务器端 IP 地址比较，如果 IP

地址不匹配，有可能发生 Cookie 被劫持，抑或是用户在同一个地点使用笔记本里用其他无线接入点上网。

时间戳是用来记录 Cookie 的创建日期和时间，这条记录有两个目的，第一是可以检测用户多长时间没有登录；第二是用来比较用户登录间隔的时间，当发现用户在很短时间内在不同 IP 地址登录时，对登录加以限制，以减少重播攻击。如图 7-3 所示。

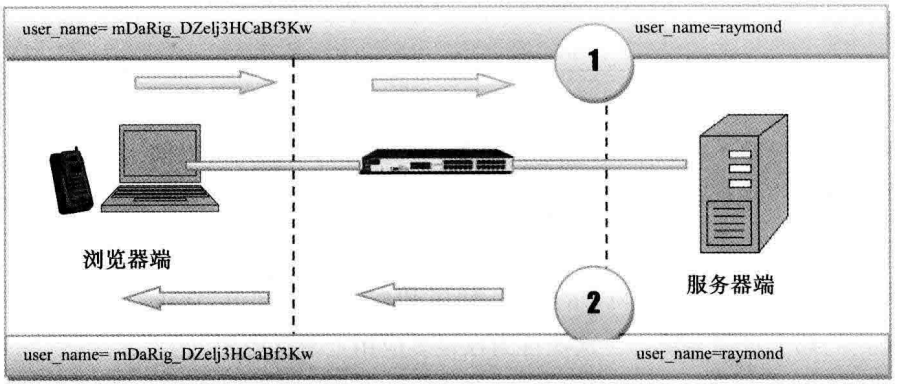


图 7-3 Cookie 加密示意图

三点信任度可以表示使用这个 Cookie 设计：

- 最信任：IP 地址与最近的时间戳匹配。
- 较信任：IP 地址匹配，但时间戳太旧，这表示用户已离开计算机，属于无人值守状态。
- 最不信任：IP 地址不匹配。

这个理想化的安全措施，使用什么样的信任程序需酌情分析。例如，一个网站仅仅实现一个欢迎页面，就没有必要使用高度信任。

当出现不信任的情况，需要提示用户再次输入密码。若用户输入密码正确，时间戳和 IP 地址将被重置。例如，用户在访问一个网上商店浏览商品并选择了一些加入购物车，当他再次访问网站，查看他的访问历史或购物车时，就需要再次输入用户名和密码。

对于涉及财务或数据安全的，就需要对 Cookie 进行加密。如下列代码清单：

```
$cookieData = serialize( $user );
$iv_size = mcrypt_get_iv_size(MCRYPT_RIJNDAEL_256, MCRYPT_MODE_CBC);
srand();
$iv = mcrypt_create_iv($iv_size, MCRYPT_RAND);
$encryptedData = mcrypt_encrypt(MCRYPT_RIJNDAEL_256, $secret,
$cookieData, MCRYPT_MODE_CBC, $iv);
```

```
setcookie( 'user', base64_encode($encryptedData).':' .$iv );
```

那么解密 Cookie 类似以下代码：

```
list($encryptedData,$iv) = explode( ':' , $_COOKIE[ 'user' ]);
$rawData = mycrypt_decrypt( MCRYPT_RIJNDAEL_256, $secret,
base64_decode($encryptedData),
MCRYPT_MODE_CBC, $iv );
$user = unserialize( $rawData );
```

其中，$secret 是加密的 key，是在应用程序端设置的。加密的 Cookie 有 3 个部分：一个私钥 key，初始化向量 $iv 和 cookie 数据本身，可以给私钥加一个 salt 串，可以是用户 ID 或者随机字符混合，这样攻击者无法破解。

作为安全的最后级别，加密有益于用户验证以及用户数据的安全，我们将 UserID 也同时添加到 Session 中，当 Cookie 和 Session 两端的 UserID 不匹配时，就可以证明有人在伪装 Cookie，或者数据被破坏，这时登录的 Session 和 Cookie 将被销毁。

如果不存在 Session，即代表本次会话结束，当会话重新开始时，需要提示更换了上网场所，需要用户重新输入密码，再开始新的会话。

注意　从可用性的角度来看：良好的软件不仅有强大的功能和安全性，也应该容易使用。由于 Cookie 中包含用户 ID 号（UserID）以及有可能确定用户的登录名称，当再次提示输入密码，会话过期后，登录名应该是自动提示并填写在表单上。

这样，我们在应用程序层保证了用户数据的安全，并有效地防止了 Cookie 欺骗，攻击者不能够轻易解密用户的信息。典型的例子有飞信和 QQ 之类的系统，就使用这样的措施，使用户的账号和数据不轻易丢失。

另外，还需要对用户提供不同级别的处理，如管理员、编辑、客服人员、普通会员等角色。

7.4　访问控制列表

访问控制列表（Access Control List，ACL）是一种细粒度的管理应用程序权限的方式，虽然粒度很细，但却很容易维护和管理。控制谁在系统的动作是什么，都有谁可以使用。

在一些内容管理系统中会使用到 ACL，一般最普遍使用的 ACL 系统包含以下两种类型：

- 组级：一个用户可以属于一个或多个组，每个组都有自己的权限，组内成员可以定义，主要是游客、会员、付费会员、版主和管理员等。
- 用户级：用户级权限始终覆盖组级别的权限。例如，在正常情况下，只有版主和管理员可以编辑论坛中的文章，但他们可以给“丁丁”权限直接编辑论坛帖子。

来看代码清单7-10：

代码清单7-10 ACL数据库表结构

```
CREATE TABLE `groups` (
id INT AUTO_INCREMENT PRIMARY KEY,
name VARCHAR(128)
);
CREATE TABLE `group_users` (
`user_id` INT,
`group_id` INT,
KEY ( `user_id`, `group_id` ),
KEY ( `group_id` )
);
CREATE TABLE `group_permissions` (
`group_id` INT,
`permission` VARCHAR(50),
KEY ( `group_id`, `permission` ),
KEY ( `permission` )
);
CREATE TABLE `user_permissions` (
`user_id` INT,
`permission` VARCHAR(50),
KEY ( `user_id`, `permission` ),
KEY ( `permission` )
);
```

我们选择指定用户的所有权限，使用联合连接两个权限表，该查询类似于以下（真实环境，请替换UserID）：

```
( SELECT `permission` FROM `group_permis
sions`
WHERE `group_id IN (
SELECT group_id FROM group_users WHERE user_id=USER_ID
))
UNION
( SELECT permission FROM user_permissions WHERE user_id=USER_ID )
```

该查询将返回字符串列表。为某个用户添加权限，类似于如下SQL语句：

```
INSERT INTO `user_permissions` (`user_id`, `permission`)
VALUES (USER_ID, "blog/add");
```

本节所讲解的是一个基本的 ACL 设计。一个完整的 ACL 系统还可以包括资源控制，另外当权限配置在数据库中保存时，如果启用缓存会导致生效延迟等问题，需要注意。

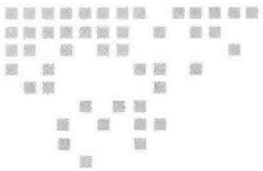
7.5　本章小结

本章涵盖了所有 PHP 中有关用户验证的方法和解决方案，包括基本的 HTTP 验证、摘要式的身份验证以及使用 PHP 的验证。

摘要式身份验证有较复杂的算法格式，因此需反复理解，亦可在需要时查阅。另外，这两种方法如果配合 SSL 方式，安全性是可圈可点的。

后面我们又了解 Session 自定义存储以及 cookie 安全处理，其中的代码均可作为产品开发时的参考，适当修改程序就可应用。

最后，我们介绍了很多应用都会用到的用户角色的权限处理方法，相信会对开发者在安全编程方面有一定借鉴与实践意义。



Chapter 8 第 8 章

深度理解 MySQL 驱动与存储引擎

当网站后端出现性能瓶颈时，大多数开发者第一反应都会去 PHP 脚本中查找问题所在，这样做当然没有任何问题，但是需要留意的是，一些问题与 MySQL 和存储引擎也存在关联。

与别的 RDBMS 相比，MySQL 是一个比较特别的数据管理系统，它通过存储引擎来让应用程序以不同的方式存取数据，它可为不同的应用提供不同的功能，因此在性能与功能上也存在些许差别。

作为开发者，需要详细了解这些功能之间的区别，比如事务和非事务支持、表级与行级锁定策略，包括数据完整性（强关联）、外键、索引类型的区分，以及 B-Tree、哈希算法和全文检索等 MySQL 引擎的特性等。

8.1 MySQL 连接驱动库

MySQL 在软件设计上采用基于组件的模块化设计，使用 C/C++ 开发。架构是一个类似于子系统组成的架构，子系统通过紧密和高效配合，组成一个可靠的数据库系统。

MySQL 数据库的内部结构可以分为几个模块，且大部分的层次还可进行拆分。大部分子系统依赖于一些通用的底层库。

图 8-1 为我们展示了 MySQL 数据库的框架和子系统。

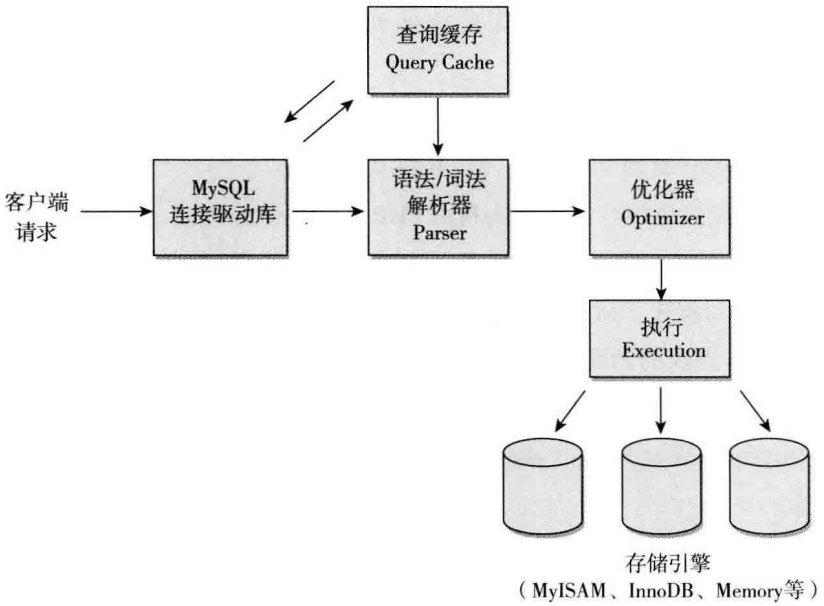


图 8-1　MySQL 数据库的框架和子系统

MySQL 包含以下子系统（模块）和核心库：

- 网络连接和网络通信协议库与子系统。
- 线程、进程和内存分配子系统。
- 查询解析和查询优化子系统。
- 存储引擎接口子系统。
- 存储引擎子系统。
- 安全管理子系统。
- 日志子系统。
- 其他子系统，如主从复制、集群、错误处理等。
- 低层核心 API（mysys/string）。

8.2　mysqlnd 驱动

PHP 从 5.3 版开始，为开发者提供了与 MySQL 数据库不同方式的连接与网络通信驱动库。在之前，PHP 编译的连接驱动库是 libmysql，它是由 C++/C 开发的扩展库，实现了与 MySQL 服务器进行底层网络通信的协议，但它存在一些较明显的缺点：

- 编译困难，在编译过程中需要连接到 MySQL 官方网站，然后再分发到当前主机。
- 与 PHP 许可证（PHP License 3.0）不兼容，libmysql 使用的是 GPL 许可证。
- 使用 libmysql 驱动的 mysqli 不支持持久化连接。
- 数据库性能虽也很好，但无法利用 PHP 本身提供的特性。

综上原因，PHP5.3 也用 C 重写了连接 MySQL 的驱动库，称为 mysqlnd，意为 MySQL 原生的驱动程序（MySQL Native Driver）。它是一个完全用 PHP 许可证编写的，遵守 MySQL 通信协议的新驱动，这意味着它和 PHP 一样方便分发。无须再去 MySQL 官网下载，再分别每台机器编译，可以不需要遵守 MySQL 的 GPL 许可证。

改进的 mysqlnd 驱动库支持 mysqli 客户端 API 库的持久化连接，mysqlnd 还有更多更好的表现：

- mysqli_fetch_all() 函数与 mysqlnd。
- 性能统计添加到 mysqli 库。

由于 mysqlnd 是纯 PHP 扩展，意味着它可以使用及管理 PHP 内存和其他扩展库，与 PHP 集成紧密，能够提供更好的内存性能和管理资源。

mysqlnd 和 Zend 引擎高度集成，充分使用了 PHP 的流 API（Stream API）以及客户端缓存机制，执行速度更快、内存消耗更少。由于 mysqlnd 和 Zend 引擎集成，因此可以提供更多高级特性，并更有效地利用 Zend 引擎进行加速，原理如图 8-2 所示。

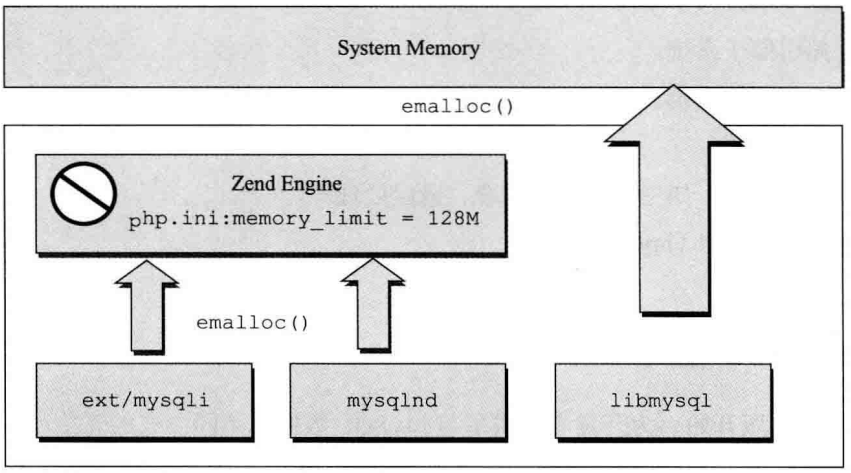


图 8-2 mysqlnd 原理

从图 8-2 可看出，libmysql 直接访问数据库，而 mysqlnd 是通过 Zend 引擎访问数据库。myslqnd 使用 Zend 引擎进行内存管理，因此它遵守 PHP（与 libmysql 相比）的内存限制设置，可分配更大的内存块（分配内存是比较困难的任务），它会尝试重用 zval，并使用 zval 的缓存，以节省 CPU 资源。这样既节省了一层函数调用和数据复制，节省了内存、提高了性能，又解决了许可证的问题。

libmysql 和 mysqlnd 这两个扩展库都使用 MySQL 的标准 API，由于 mysqlnd 提供更新的功能，我们将选择 mysqlnd，mysqlnd 编译安装很方便。

我们在编译 PHP 时，使用 libmysql 的编译命令是这样的：

```
./configure --prefix=/usr/local/php \
            --with-mysql=/usr/local/mysql \
            --with-mysqli=/usr/local/mysql/bin/mysql_config \
            --with-pdo-mysql=/usr/local/mysql
```

换成 mysqlnd，编译命令如下：

```
./configure --prefix=/usr/local/php \
            --with-mysql=mysqlnd \
            --with-mysqli=mysqlnd \
            --with-pdo-mysql=mysqlnd
```

编译时不用指定 mysql 的位置信息，仅提供一个 mysqlnd 参数值即可。待编译完成就可以使用 mysqlnd 作为 MySQL 的连接和通信驱动库，它将具有原生 MySQL 驱动的所有特性并且可以提供更好的功能，具有更高的可维护性。想要验证查看当前的连接驱动是否为 mysqlnd，可以使用如下代码：

```
<?php
$mysqlnd = function_exists('mysqli_fetch_all');
if ($mysqlnd) {
    echo 'mysqlnd enabled!';
}
```

要测试 PDO 中使用的 mysqlnd 驱动是否正常，可用如下代码：

```
if (strpos($pdo->getAttribute(PDO::ATTR_CLIENT_VERSION), 'mysqlnd') !== false) {
    echo 'PDO MySQLnd enabled!';
}
```

接下来我们来看 mysqlnd 到底给开发者带来了哪些更具体的增益。

1. 节省内存

mysqlnd 一个有趣的概念称为“只读变量”，它可以节省大量的服务器内存。如图

8-3 所示，当你运行一个查询缓存时，libmysql 读取数据到自己的缓冲区，然后当运行 mysql_fetch_*() 函数时会将数据从该区复制到自己的存储区。而 mysqlnd 会动用只读缓冲区，但是当运行 mysql_fetch_*() 函数时，不会去复制数据，而是像 C 中的指针一样，将 PHP 变量链接到这个内存区域，这种处理方式只有 libmysql 库内存消耗的一半。

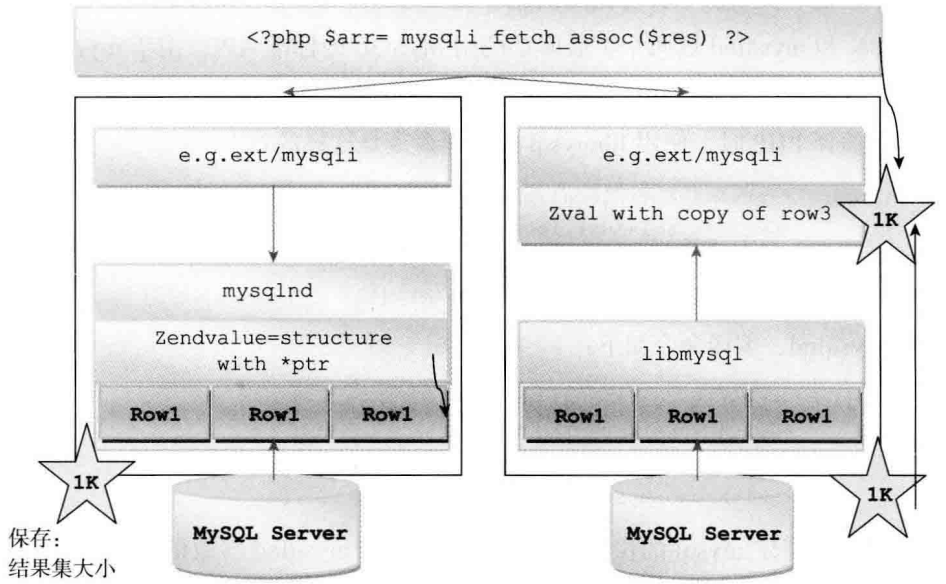


图 8-3 mysqlnd 运行过程

2. 流 API

mysqlnd 使用 PHP 流 API（Stream API），它提供大量的可用特性，使用 hook 机制与 MySQL 服务器通信。如图 8-4 所示。

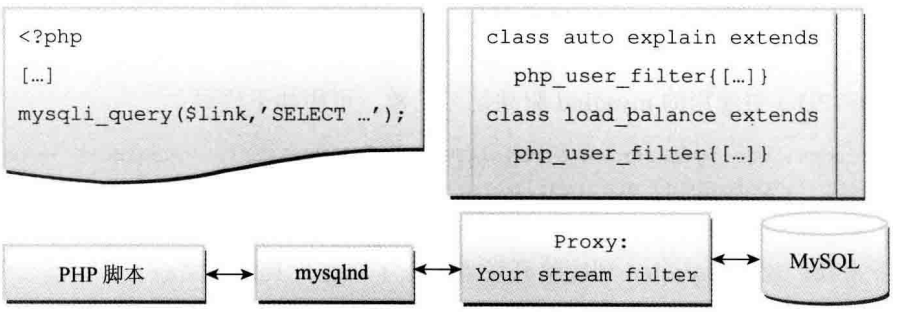


图 8-4 流 API

3. 持久化连接

以前，MySQL 扩展中不允许使用 libmysql 的持久化连接功能，而使用 mysqlnd 将可再次使用持久连接，如 mysqli_pconnect(p:localhost',...);。

4. 静态数据统计

mysqlnd 帮助开发者收集了大量的统计数据，用于调整应用程序使用。通过这些信息很容易找出系统中的瓶颈，利用这一特性可轻松地监视应用，甚至可以自动检测和修复问题。

可通过 phpinfo() 来查看 mysqlnd 提供的函数，比如客户端统计函数 mysqli_get_client_stats()，连接数统计函数 mysqli_get_connection_stats() 等。

mysqlnd 目前提供了近 59 个统计函数。比较有用的是取得 zval 缓存内部状态信息的函数 mysqli_get_cache_stat() 等。

5. 客户端查询缓存

mysqlnd 支持客户端查询缓存，这在某些情况下很有用，它可节省带宽（网络延迟）和内存分配开销。通过客户端查询缓存，它可能是 mysqlnd 的无效缓存，mysqlnd 目前只支持 TTL 过期（生存时间），这可以通过 php.ini 设置控制。

这部分目前是试验性的，在未来可能发生较大的改变。

6. 未来的功能增强与新特性

在这方面有很多的可能性，包括改善客户端的查询缓存、获取 PHP 流数据给用户、支持 prepare 语句、高速缓存、自动负载平衡、内置的分析工具等。

8.3　存储引擎

MySQL 提供一个独特的架构，数据管理上提供了不同的存储引擎定义持久性与信息不同特性的查询。像我们每个人一样，每个存储引擎都有自己的优势和劣势，因此选择一个单一的存储引擎来完成任务是完全理想化的。MySQL 允许开发者在一个数据库模式中混合使用多种存储引擎，这样显然会提高复杂性，如事务支持和备份策略的执行与功能都要经过深入思考。

MySQL 从 5.1 版本开始使用插件式存储引擎体系架构（Pluggable Storage Engine Architect，PSEA），这样可以让存储引擎层与 SQL 层相对独立，耦合变小，实现动态加

载环境，使存储引擎的加载移除变得方便，开发者自行开发存储引擎变得容易。这种思想也将数据库理论引用到产品中来，完美地映射了数据库的外模式和内模式理论。

MySQL 的存储引擎主要包括 MyISAM、InnoDB 以及其他引擎，这些引擎相对来说都应用于特定的场合。存储引擎的共通技术特性包括如下：

- 事务性和非事务的持久性。
- 非持久性。
- 表锁定和行锁定。
- 不同的索引算法，如 B-Tree/B+Tree 索引、R-Tree 索引，Full-text 全文检索。
- 群集索引，主从库。
- 数据压缩。
- 全文检索。

在写作时，Oracle 发布了 MySQL 版本是 5.6.10 GA，这一版本主要是修复了之前 RC 版本中的 Bug。其新增特性包括如下：

- InnoDB 存储引擎改进，增加全文索引功能。这一特性让我们完全可以使用 InnoDB 来代替 MyISAM。
- 提升子查询性能。子查询的性能问题一直是困惑程序员们的问题，为避免这一问题，我们不得不分成两步查询来解决，这下好了，直接用子查询就可以。
- 同步复制功能增强，引入多线程复制特性。
- 引入 NoSQL 特性，可以直接使用 Memcached API 操作 InnoDB 数据库。

8.3.1 取得存储引擎信息

可以使用 SHOW ENGINES 命令和查看 INFORMATION_SCHEMA 表来查看当前 MySQL 提供了哪些存储引擎。

在 MySQL 命令行下运行 SHOW ENGINES 命令，来查看引擎支持的详情，可以看到存储引擎是支持事务性还是非事务。如图 8-5 所示。

8.3.2 定义存储引擎

从 MySQL5.0 开始，默认的存储引擎是 MyISAM，它是之前版本 IASM 存储引擎的升级版本，是目前应用最广泛的引擎。

我们可以修改 MySQL 的配置文件 my.cnf，更换系统的默认存储引擎，还可以在

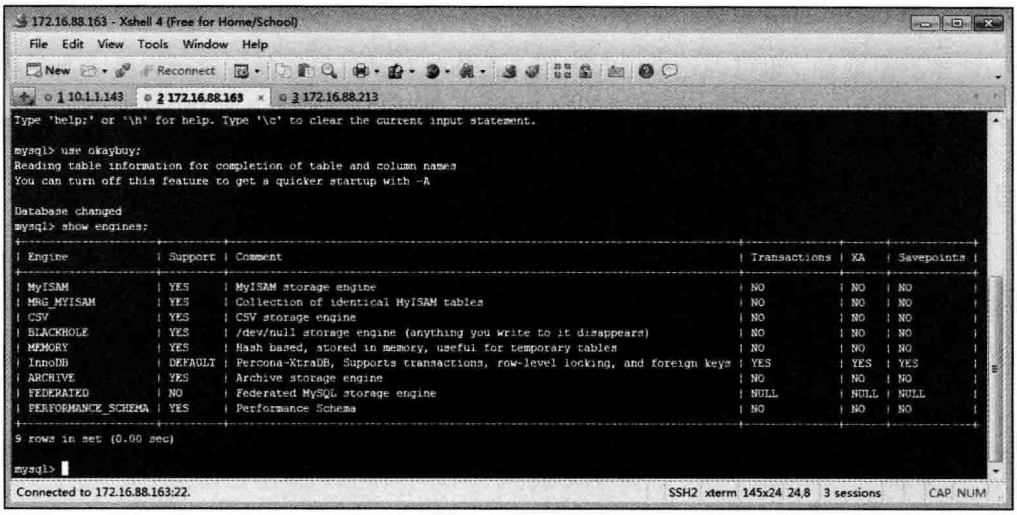


图 8-5　查看引擎支持

CREATE TABLE 或 ALTER TABLE 命令后面定义数据存储引擎。它可以在字段定义的后面，也可以在表定义的后面。

创建一个 MyISAM 存储引擎的数据表：

```
mysql>CREATE TABLE example_myisam(
> id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
> c VARCHAR(100) NOT NULL)
> ENGINE=MyISAM;
mysql>SHOW CREATE TABLE example_myisam
```

创建一个内存表：

```
mysql>CREATE TABLE example_memory(
> id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
> c VARCHAR(100) NOT NULL)
> ENGINE=MEMORY;
mysql>SHOW CREATE TABLE example_memory
```

如果在 CREATE TABLE 命令后面不指定存储引擎，默认使用 my.cnf 或者系统默认的存储引擎。MySQL 参数调整大多数通过配置文件 my.cnf，在 Linux 下通常保存在 / etc 或 / etc / mysql 目录中。MySQL 没有提供一个配置重载功能，如果在 my.cnf 做出了任何修改，需要重新启动来让它重新读取配置。

还有一种替代方案，如果我们有 Shell 权限，也可以在 MySQL 命令行（CLI）下设置全局变量（那些出现在 my.cnf 中的参数）。语法如下：

```
set variable_name = value;
```

这样不需重启，也可以让设置起作用，在临时测试时可以使用这种方法。当服务器重启时，设置将会消失，如果要设置持久起作用，还是老老实修改配置文件吧。

8.3.3 内置的存储引擎

存储引擎是MySQL的特有属性，我们可以自由选择一个内置的引擎（甚至可以自己编写）。本节将在性能方面介绍两个主要的存储引擎（MyISAM和InnoDB），并涵盖另外一些常见的存储引擎。默认情况下，MySQL的官方二进制包中包含内置存储引擎。

1. MyISAM

MyISAM是MySQL 5.1的默认存储引擎。该存储引擎的核心功能如下：

- 插入速度快。
- 使用B tree索引。
- 支持FUULTEXT全文检索。
- 16K数据页。
- 4K索引面。

限制和缺点：

- 不支持事务。
- 不支持外键。

重要参数：

- key_buffer_size：这个缓冲区是用来保存的MyISAM表的索引数据。MyISAM数据可以支持多个索引缓存定义，也可以保存特定缓冲区中特定表的索引。
- table_cache：打开表运行查询时所使用的缓冲区保存的信息。虽然适用于所有存储引擎，但由于额外的网络连接LES，它是重要的调整参数时，有很多的表和复杂的查询。
- bulk_insert_buffer_size：这个缓冲区用来改善INSERT语句大量的值或INSERT…SELECT以及LOAD DATA INFILE。
- myisam_recover：此参数定义MyISAM存储引擎的默认恢复模式。

一个MyISAM表在文件系统分为3个独立的文件，这些文件分别是：

- table.frm：MyISAM表定义文件。
- table.myd：MyISAM数据文件。

- table.myi：MyISAM 索引文件。

MyISAM 表支持 3 种类型的索引：

- B-tree 索引。所有的索引节点都按照 balance tree 的数据结构来存储，所有的索引数据节点都在叶节点。
- R-Tree 索引。与 B-Tree 区别是，主要设计用于为存储空间和多维数据的字段做索引。
- Full-text 全文检索。全文检索，它的存储结构为 B-tree，主要是为了解决需要用 LIKE 查询时的低效问题。

MyISAM 是在读频率相对高的环境，或者高写入环境的最佳存储引擎，但不适合读 / 写都很高的环境。主要问题是，DML 语句会导致表锁定，如果发生这种情况，所有待读取的进程将被阻塞，直到 DML 语句执行完成。

我们可以通过运行一个基准测试，比如执行大量重复的 SELECT 语句，然后使用 SHOW PROCESSLIST，你会看到一个被锁定的状态。在使用一个 UPDATE 语句时，例如，需要时间来执行，也会看到显示 SELECT 语句被锁定的状态。

另外使用 MyISAM 最大的问题是缺乏数据完整性，MyISAM 在达到高频的写入吞吐量时，会导致 MySQL 崩溃，比如作为 Session 会话时的 MyISAM 表会经常死掉。

因为 MyISAM 没有事务支持，并因此缺乏回滚的一致性，在 DML 语句中缺少索引数据的同步。比如在一个 INSERT 语句，数据被写入磁盘，但查询时该数据可能不在，它会按 key_buffer 的时间写入磁盘，因此产生查询不同步。

在服务器崩溃的情况下，数据库文件出现不一致时，可使用 REPAIR TABLE 命令来尝试恢复。这里还有一些问题：第一，我们一般不知道损害的程度，直到表和索引访问。虽然 mySQL 提供了这个检查 TABLE 命令，再无其他工具。第二，恢复时间依赖于数据库的大小。数据库越大，恢复的时间也会随之增长，这种问题经常重现，我们无法预料，只能定期进行常规修复。在出现问题后，时间可能消耗较大，及时性、可用性被降低。

在 MySQL 停止的情况下，可以使用 MyISAM 工具对某个表进行检测或修复。

2. InnoDB

在 MySQL5.5 之后，InnoDB 存储引擎成为默认的存储引擎，主要的原因是完整的事务支持。其关键技术特性如下：

- 安全事务支持。

- 全 ACID 支持。
- 支持 MVCC（Multi Version Concurrency Control，多版本并发控制）。
- 行级锁定。改变了 MyISAM 的锁定机制，InnoDB 实现了行级锁定。
- 外键支持。与 MyISAM 不同，InnoDB 实现支持外键引用。
- 主键支持使用集群索引。

默认情况下在数据库目录中，InnoDB 分为两个单独的文件。这些文件分别是：

- table.frm：数据表定义文件。
- ibdata1：默认的 InnoDB 所有表空间。

此外，InnoDB 一般有两个事务日志文件，分别是：

- ib_logfile0。
- ib_logfile1。

可以通过修改配置来更改 InnoDB 表空间和日志文件的默认名称与路径。

作为开发者，我们应熟悉 MyISAM 和 InnoDB 之间的具体差异。如前所述，使用 InnoDB 的首要原因是事务的支持。还有一个显著差距就是行级锁定。

MyISAM 存储引擎速度虽然非常快，但它在高写入和读取环境时的表锁定问题仍然难办，能克服的方法就是升迁存储引擎为 InnoDB。

然而，任何改变策略都会影响应用程序的功能和性能。我们需要考虑的几个主要因素如下所示：

- InnoDB 会增加磁盘占用率。InnoDB 对主键重新设计，因此磁盘空间的使用是 MyISAM 的 3 倍以上。
- COUNT(*) 性能低下。在 MyISAM 引擎由于有个内置的计数器执行 count(*) 速度非常快，而且执行速度与记录条数无关，而 InnoDB 却不是这样，记录越多速度会变慢，这一问题会在未来版本中会有所改观。
- 不支持 FULLTEXT 全文检索。在 MySQL5.5 之前不支持，从 5.6 版本后已经完全支持该特性，如果你使用了最新版本，可以忽略此差异。
- 在 SQL 查询执行的差异（QEP）。

3. Memory

这个存储引擎在之前被称为 HEAP 表，从 MySQL 5.0 开始被正名为 Memory。Memory 表使用哈希散列索引把数据保存在内存中，因此具有极快的存取速度，适合缓存中小型数据。它在使用上受到一些限制：

- Memory 存储引擎不会将任何数据存储到硬盘上，在磁盘上会存储一个与表结构相关的 .frm 文件。因此，一旦 MySQL 崩溃后，Memory 就剩下一个表结构。
- Memory 存储引擎支持索引，并且同时支持 Hash 和 B-Tree 两种格式的索引，前者是默认值。
- 由于是放在内存中，它按照定长空间来存储数据，不支持 BLOG 和 TEXT 类型字段。
- Memory 存储引擎实现的是页级锁定。
- max_heap_table_size 都可变，Memory 表的大小是有限的，默认为 16 MB。

最后，网络连接存储引擎使用固定的大小分配的内存中的每一行是行，确定的行的大小。这是有点低效，并且可以导致在一个特定内存被浪费。

关于 B-Tree 与哈希索引

Memory 是所有存储引擎中提供 B-Tree 和哈希索引两种选择的存储引擎。MyISAM 和 InnoDB 存储引擎仅使用了 B-Tree，哪种更好？

B-Tree 特别适合大的索引，尽可能是让内存中为一个整体结构。他们考虑到大部分的 Tree 被存储在二级存储中（即磁盘中），被设计为最大限度地减少磁盘读取次数。

由于树结构，B-Tree 可以很容易地执行范围式查询，比如 =,BETWEEN 以及相似度查询，如 LIKE '爱 %'。

哈希索引相比较 B-Tree 来讲没有后者灵活，但速度更快。它不会去遍历整个树，MySQL 可以直接去数据表中取值，这种表的结构也会导致没有特别有效的范围查询。使用 B-Tree，一个分支可以被返回，而使用哈希表，需要对表整体进行扫描。

因此，B-Tree 通常应用于大的索引（不会断断续续地保存在内存中）。哈希表速度很快，适合于规模较小的索引（不扩展为 B-Tree），并且仅仅用于测试环境。

4. Blackhole

这个存储引擎很特别，直译为中文的意思就是“黑洞”，就如同 Linux 下使用 /dev/null 设备一样，不管我们写入什么信息都是有去无回。

有人使用 Blackhole 存储引擎实现 mysql replication 的部分复制，在数据迁移过程中，数据需经过中转服务器进行转换操作，然后再复制移植到新的服务器上面。

如果中转服务器没有相应的空间来支持这种操作，就可以使用 Blackhole 功能，虽然它不会记录任何数据，但在 MySQL Bin Log 中会记录所有的查询，可以充分利用这

些查询来复制利用，然后复制到转换端，如图 8-6 所示。下面给出一个链接，里面详细地解释了这是怎样实现的：http://jroller.com/dschneller/entry/mysql_replication_using_blackhole_engine。

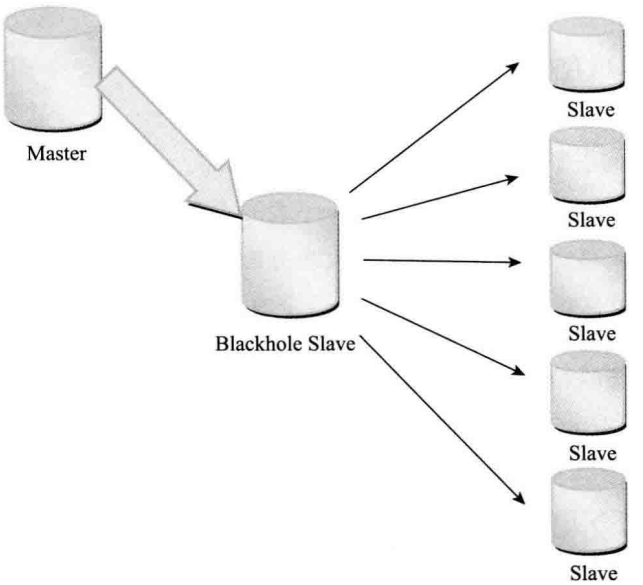


图 8-6 Blackhole

5. Archive

有时开发者或 DBA 管理着海量数据库的人常常要回答诸如“何人何时修改了什么”或者“何人何时查看了什么”这样类似的问题。比如像数以千计的员工，开展着不计其数业务的企业，每天都会产生出大量的日志记录数据，而且必须严格保存。

为了帮助应对数据爆炸的挑战，MySQL5.0 引入了一种新的数据存储引擎，叫作 Archive。这个先进的数据管理工具，让我们拥有了处理和管理海量数据的新式武器。Archive 引擎为大量很少引用的历史、归档或安全审计信息的存储和检索提供了完美的解决方案，因此不提供索引功能。

Archive 使用 Zlib 算法压缩，比 MyISAM 表要小大约 75%，比 InnoDB 表小约 83%。当数据量非常大的时候 Archive 的插入性能表现会较 MyISAM 佳。

Archive 表的性能是否可能超过 MyISAM？根据 MySQL 工程师的资料，当表内的数据达到 1.5GB 这个量级，CPU 又比较快的时候，Archive 表的执行性能就会超越 MyISAM 表。因为这个时候，CPU 会取代 I/O 子系统成为性能“瓶颈”。

Archive 表比其他任何类型的表执行的物理 I/O 操作都要少。较小的空间占用也能在移植 MySQL 数据的时候发挥作用。当你需要把数据从一台 MySQL 服务器转移到另一台的时候，Archive 表可以方便地移植到新的 MySQL 环境，只需将保存 Archive 表的底层文件复制过去就可以。

6. CSV

CSV，是 Comma Separated Value（逗号分隔值）的英文缩写，通常为纯文本文件，文件中的字段以逗号分隔。MySQL 的 CSV 存储引擎也是保存这样格式的数据（在 Windows 下该存储引擎不可用）。CSV 存储引擎包含在 MySQL 二进制分发包，想要开启此存储引擎，需要在编译时加入 --with-csv-storage-engine 选项。

当创建完 CSV 表后，会生成 3 个文件：

- csv_table.frm。这个文件保存表的元数据信息，任何一个 MySQL 存储引擎都有这个表文件。
- csv_table.csv。这个是数据文件，保存表中的数据，用 CSV 格式保存。
- csv_table.csm。相关的资源文件。

使用 CSV 存储引擎的一个重要好处是，可以直接通过修改文件，向数据库中添加更新数据。MySQL 在运行期间锁定了该文件，因此必须停止了 MySQL 才能使用其他工具修改。CSV 存储引擎实际上操作维护的是一个标准的 CSV 文件，因此并不支持索引。

下面介绍几种其他引擎。

1. Federated

本地 MySQL 数据库要访问远程 MySQL 数据库的表中数据，可以通过 Federated 存储引擎来实现。有点类似 Oracle 中的数据库链接（DBLINK）。要开启 Federated 存储引擎，在编译 MySQL 时要使用 --with-federated-storage-engine 选项。

当创建一个 Federated 表时，和其他存储引擎一样，在数据库目录创建一个表定义文件，再无其他文件被创建，因为实际的数据在一个远程数据库上。

使用 Federated 表很方便。通常运行两个服务器，要么在同一个主机上，要么在不同主机上。

2. BDB

BDB 引擎全称为 Berkeley DB。对 BDB 存储引擎的支持包含在 MySQL 源码分发版里，在 MySQL-Max 二进制分发版里需激活。BDB 数据库引擎在收购 Sun 之前就已被

Oracle收购，因此名正言顺将其纳入MySQL数据库的产品线。

如果想要使用BDB引擎，需要在编译时加入./configure --with-berkeley-db=./bdb选项。

BDB被设计成可以替代InnoDB的事务引擎，支持COMMIT、ROLLBACK和其他安全事务特性。BDB被创建后，包括两个物理表文件：frm与.db文件，数据和索引均放在.db文件里。

3. NDB

NDB存储引擎也叫NDB Cluster，用于MySQL分布集群环境，类似于Oracle的RAC集群，与其结构不同的是，其结构也是Share Nothing的集群架构，因此能提供更高级别的高可用性。

NDB的特点是数据全部放在内存中（从5.1版本开始，可以将非索引数据放在磁盘上），因此主键查找（primary key lookup）的速度极快，并且通过添加NDB数据存储节点（Data Node）可以线性地提高数据库性能，是高可用、高性能的集群系统。

关于NDB存储引擎，有一个问题值得注意，那就是NDB存储引擎的连接操作（JOIN）是在MySQL数据库层完成的，而不是在存储引擎层完成的。这意味着，复杂的连接操作需要巨大的网络开销，因此查询速度很慢。如果解决了这个问题，NDB存储引擎的市场应该是非常巨大的。

简单地说，MySQL Cluster就是在无共享存储设备的情况下实现的一种内存数据库Cluster环境，主要是通过NDB Cluster存储引擎来实现的。一般来说，一个MySQL Cluster的环境主要由以下三部分组成。

（1）负责管理各个节点的Manage节点主机。

管理节点负责整个Cluster集群各个节点的管理工作，包括集群的配置、启动关闭各节点，以及实施数据的备份恢复等。管理节点会获取整个Cluster环境中各节点的状态和错误信息，并且将各Cluster集群中各个节点的信息反馈给整个集群中其他的所有节点。由于管理节点上保存着整个Cluster环境的配置，同时担任子集群中各节点的基本沟通工作，所以它必须是最先被启动的节点。

（2）SQL层的SQL服务器节点，也就是我们常说的MySQL Server。

MySQL Server主要负责实现一个数据库在存储层之上的所有事情，比如连接管理、query优化和响应、cache管理等，只有存储层的工作会交给NDB数据节点去处理。也就是说，在纯粹的MySQL Cluster环境中的SQL节点，可以被认为是一个不需要提供任

何存储引擎的 SQL 服务器，因为它的存储引擎由 Cluster 环境中的 NDB 节点来担任。所以，SQL 层各 SQL 服务器的启动与普通的 MySQL 启动有一定的区别，必须添加 NDB Cluster 项，可以添加在 my.cnf 配置文件中，也可以通过启动命令行来指定。

（3）Storage 层的 NDB 数据节点，也就是上面说的 NDB Cluster。

NDB 是一个内存式存储引擎，也就是说，它会将所有的数据和索引数据都加载到内存中，但也会将数据持久化到存储设备上。不过，最新版本已经支持非索引字段数据不用全部加载到内存种类，这对于有些数据量太大或基于成本考虑而没有足够内存空间来存放所有数据的用户来说，的确是一个大好消息。

NDB 节点主要是实现底层数据存储的功能、保存 Cluster 的数据。每一个 NDB 节点保存完整数据的一部分（或者一份完整的数据，视节点数目和配置而定），在 MySQL Cluster 里面叫作 fragment。而每一个 fragment，正常情况来讲都会在其他的主机上面有一份（或者多份）完全相同的镜像存在。这些都是通过配置来完成的，所以只要配置得当，MySQL Cluster 在存储层就不会出现单点的问题。一般来说，NDB 节点被组织成一个一个的 NDB Group，一个 NDB Group 实际上就是一组存有完全相同的物理数据的 NDB 节点群。

8.4　第三方存储引擎

第三方存储引擎，即非 MySQL 团队开发提供的存储引擎，一般是由在某些方面提供特性的软件厂商研发的。

1. TokuDB

TokuDB 是一个应用在 MySQL 和 MariaDB 中的存储引擎，它使用索引加快查询速度，具有高扩展性，并支持 Scheme 热修改。

特点：

- 插入性能快 20~80 倍。
- 压缩数据，减少存储空间。
- 数据量可以扩展到几个 TB。
- 不会产生索引碎片。
- 支持 hot column addition、hot indexing、mvcc。

如何考虑使用 TokuDB：

- 如果要存储blob，不要使用TokuDB，因为它的记录不能太大。
- 如果记录数过亿，使用TokuDB。
- 如果注重update的性能，不要使用TokuDB，它的速度没有InnoDB快。
- 如果要存储旧的记录，使用TokuDB。
- 如果要缩小数据占用的存储空间，使用TokuDB。

2. Infobright

Infobright是一个与MySQL集成的开源数据仓库（Data Warehouse）软件，可作为MySQL的一个存储引擎来使用，查询与普通MySQL引擎并无区别。Infobright的基本特征有哪些呢？我们列举如下。

优点：

查询性能高：百万、千万、亿级记录数条件下，同等的SELECT查询语句，速度比MyISAM、InnoDB等普通的MySQL存储引擎快5～60倍。

存储数据量大：TB级数据大小，几十亿条记录。

高压缩比：极大地节省了数据存储空间。

基于列存储：无须建索引，无须分区。

适合复杂的分析性SQL查询：SUM, COUNT, AVG, GROUP BY。

限制：

不支持数据更新：社区版的Infobright只能使用“LOAD DATA INFILE”的方式导入数据，不支持INSERT、UPDATE、DELETE。

不支持高并发：只能支持10多个并发查询。

8.5 结合硬件的引擎

1. Kickfire

让每一个MySQL数据库服务器都配备一个SQL芯片，这是Kickfire公司开始将目光投向数据仓库市场的一个愿景。这款SQL芯片能够直接从内存而不需要从寄存器或缓存抽取数据，所以可以缓解导致数据查询缓慢的I/O“瓶颈”。

通过采用Kickfire的技术，可以将一个SQL查询分割成并行查询计划，将其传送到该SQL芯片，使其能够并行处理数据。当经过查询的数据以压缩格式从内存返回后，这些数据会流入到SQL芯片并在流入的过程中被加工处理。

SQL 芯片接入到现有硬件的方式类似于图形芯片连接到主服务器的方式。SQL 芯片就像是主服务器上的副处理器一样。SQL 芯片内置了并行性，使应用软件能够满足多 CPU 的负载速度。此外，Kickfire 为 MySQL 提供了增量负载功能，使其能够跟踪源数据库的变化，然后自动将这些变化传递给 Kickfire 应用软件。

2. Virident

如今，大部分互联网选择的服务传输架构环境都受到功率和 DRAM 成本曲线，以及硬盘响应速度慢的限制。

Virident 技术与 Spansion MirrorBit Eclipse 闪存共同创造了在主流服务器中存储更大数据集，提供强大的系统吞吐量，同时符合互联网数据中心对功耗和冷却限制的标准。

8.6　MySQL 替代品与分支

前面提到过，MySQL 于 2009 年随 Sun 公司一起被 Oracle 收购。随着相应的核心创始人和开发人员离开，分别又开发了不同的存储引擎和分支，以便在已经成为 Oracle 旗下的小海豚会收费或停止开发时，给开发者们提供 MySQL 同样的功能和特性，甚至更好的性能。

1. Percona

Percona 为 MySQL 服务器进行了改进，在功能和性能上较 MySQL 有着很显著的提升。该版本提升了在高负载情况下的 InnoDB 的性能、为 DBA 提供一些非常有用的性能诊断工具；另外有更多的参数和命令来控制服务器行为。

- SSD 设备专门优化。
- Flashcache 有 SQL 层接口。
- 允许 XtraDB 静态编译。
- 支持多种页大小。
- 提供额外的监控参数。
- 被严格的生产环境考验过。

Percona Server 只包含 MySQL 的服务器版，并没有提供相应对 MySQL 的 Connector 和 GUI 工具进行改进。

Percona Server 使用了一些 google-mysql-tools, Proven Scaling, Open Query 对 MySQL 进行改造。

2. MariaDB

MariaDB 基于事务的 Maria 存储引擎，替换了 MySQL 的 MyISAM 存储引擎，它使用了 Percona 的 XtraDB, InnoDB 的变体。这个版本还包括 PrimeBase XT (PBXT) 和 FederatedX 两个类似的存储引擎。

与 MySQL 相比较，MariaDB 更强的地方在于：

- Maria 存储引擎。
- PBXT 存储引擎。
- XtraDB 存储引擎。
- FederatedX 存储引擎。
- 更快的复制查询处理。
- 线程池。
- 更少的警告和 bug。
- 运行速度更快。
- 更多的扩展。
- 更好的功能测试。
- 数据表消除。
- 慢查询日志的扩展统计。
- 支持对 Unicode 的排序。

MariaDB 是 MySQL 分支版本，是由原来 MySQL 的作者 Michael Widenius 创办的公司所开发的免费开源的数据库服务器，但是提供了更多底层代码更改，试图提供比标准 MySQL 更多的性能改进。

此外，MariaDB 提供了 MySQL 的标准存储引擎，即 MyISAM 和 InnoDB。因此，实际上，可以将它视为 MySQL 的扩展集，它不仅提供 MySQL 提供的所有功能，还提供其他 MySQL 未提供的功能。MariaDB 还声称自己是 MySQL 的替代，因此从 MySQL 切换到 MariaDB 时，无须更改任何基本代码即可安装。

最后可能也是最重要的一点是，MariaDB 的主要创建者是 Monty Widenius，也是 MySQL 的初始创建者，他的第三个孩子的名字就叫作 Maria，Monty 成立了一家名为 Monty Program 的公司来管理 MariaDB 的开发，这家公司雇用开发人员来编写和改进 MariaDB 产品。这既是一件好事，也是一件坏事：有利的一面在于他们是 Maria 功能和 bug 修复的佼佼者，但公司不是以赢利为目的，而是由产品驱动的，这可能会带来问题，

因为没有赢利的公司不一定能长久维持下去。

MariaDB 跟 MySQL 在绝大多数方面是兼容的，对于开发者来说，几乎感觉不到任何不同。目前 MariaDB 是发展最快的 MySQL 分支版本，新版本发布速度已经超过了 Oracle 官方的 MySQL 版本。

在 Oracle 控制下的 MySQL 开发，有两个主要问题：

（1）MySQL 核心开发团队是封闭的，完全没有 Oracle 之外的成员参加。很多高手即使有心做贡献，也没办法做到。

（2）MySQL 新版本的发布速度，在 Oracle 收购 Sun 之后大为减缓。另外有很多 bugfix 和新的 feature，都没有及时加入发布版本之中。

以上这两个问题，导致了各个大公司都开发了自己定制的 MySQL 版本，包括 Yahoo!/Facebook/Google/ 阿里巴巴＋淘宝网等。MySQL 是开源社区的资产，任何个人和组织都无权据为己有。为了依靠广大 MySQL 社区的力量来更快速地发展 MySQL，另外开分支是必需的。

MariaDB 默认的存储引擎是 Aria，不是 MyISAM。Aria 可以支持事务，但是默认情况下没有打开事务支持，因为事务支持对性能会有影响。可以通过以下语句，转换为支持事务的 MAria 引擎。

```
ALTER TABLE tablename ENGINE=MARIA TRANSACTIONAL=1;
```

MariaDB 最新稳定版本为 5.5.29，开发版本为 10.0.1 Alpha。MariaDB 10.0 依然基于 MySQL 5.5 开发，但会引入 MySQL 5.6 部分特性。MariaDB 提供以下特性：

XtraDB 存储引擎替换 InnoDB，XtraDB 是 Percona 开发维护的 InnoDB 威力加强版，整合 Google、Facebook 等公司和 MySQL 社区的补丁。

- Aria 存储引擎和 Sphinx 存储引擎。
- 基于 Gelera Cluster 的 MariaDB 集群方案。
- 多主复制（将在 MariaDB 10.0 实现，由淘宝贡献）。
- Cassandra 存储引擎（将在 MariaDB 10.0 实现）。

值得一提的是，各大 Linux 发行版的 MySQL 逃亡潮合唱愈演愈烈，继 Mageia 2（原 Mandriva 社区衍生版）和 OpenSUSE 12.3 以后，Fedora 社区宣布将会在即将发布跳票的 Fedora 19 使用 MariaDB 替代 MySQL。MariaDB 是原 MySQL 创始人 Michael Monty Widenius 创建的一个 MySQL 社区分支，为避免 MySQL 落入 Oracle 收购后存在的闭源风险，同时提供更多特性及更强的性能。

3. Drizzle

Drizzle是从2008年Sun时代开始开发的产品，目标是一个更小更轻更快的数据库，用作云计算与Web应用的基础回顾。

它与MySQL6.0相同的功能有存储过程、查询缓存以及触发器，不同于MySQL的是，它使用微内核并实现了完全模块化，具备更多插件式API，如认证和日志功能，支持多核SMP优化，采用相对较少的数据类型和存储引擎。

Drizzle是MySQL6.0的一个分支，但与MySQL有很大差别，甚至声称不是MySQL的替代产品。他们期望对MySQL进行一些重大更改，想要提供一种出色的解决方案来解决高可用性问题，即使这意味着要更改我们已经习惯了的MySQL的各个方面。

Drizzle强调其基本目标是：他们不满意MySQL 4.1版本之后对MySQL代码进行的一些更改，声称许多开发人员不想花费额外的钱。他们承认其产品与SQL关系数据库甚至是不兼容的。这确实与MySQL有很大的不同。

与习惯的MySQL有如此大的变化，我们为什么还要考虑这款产品呢？准确地讲，原因与上面的是相同的，Drizzle是MySQL引擎的一次重大修改，它清除了一些表现不佳和不必要的功能，将很多代码重写，对它们进行了优化，甚至将所用语言从C换成了C++，以获得所需的代码。此外，Drizzle并没有就此结束修改，该产品在设计时就考虑到了其目标市场，即具有大量内容的多核服务器、运行Linux的64位机器、云计算中使用的服务器、托管网站的服务器和每分钟接收数以万计点击率的服务器。这是一个相当具体的市场。它太具体了吗？请记住这些类型的公司目前在其数据库方面投入的资金，如果他们安装Drizzle而不是MySQL，那么他们的服务器成本将削减一半，可以节省大量资金。

那么，是不是所有人都应该使用Drizzle呢？等等，正如Drizzle反复指出的那样，它与MySQL不兼容。因此，如果你现在使用的是MySQL平台，那么需要重写大量代码，才能使Drizzle在你的环境中正常工作。

尽管需要额外的工作才能让它运行，但它并不像Percona或MariaDB那样快速且易于使用。我之所以介绍Drizzle，是因为尽管目前它可能不是你的选择，但几年之后，它很可能会成为一些人的选择。

8.7 本章小结

由于MySQL存储引擎的灵活性，越发显得一个开发者的责任重大，应该能够确保

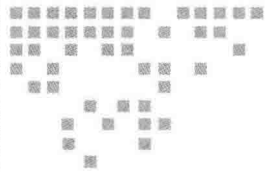
最佳的解决方案应对正在开发的产品应用。如果开发者了解通透，会很容易找到安装 MySQL 的错误配置而导致性能差的局限性。了解正确的存储引擎配置信息或存储引擎的局限性，会了解 MySQL 产品的相对优势，与应用的需求结合更紧密。

虽然 MySQL 允许混合不同的存储引擎在一个 MySQL 实例，不过仍然需要我们小心求证，包括事务支持，备份的一致性、复杂性等问题。我们决定 MySQL 启动的默认存储引擎，通常是一个事务性存储引擎或一个非事务性存储引擎，这会影响开发的方式，一般默认情况下选择 InnoDB 或 MyISAM 其一就可以。

另外我们还了解了 MyISAM 与 InnoDB 之前的差异，也会关注 MySQL 未来与 MariaDB 发展的此长彼消，另外还了解到目前其他创新的存储引擎。我们需要认真考虑选择一个存储引擎的目标，比如使用 Archive 作为数据仓库。

互联网技术变化是飞快的，有可能你在看本节时，又有新的产品出现了。

我们在本节了解到除了 MySQL 还有其他分支的 RDBMS 软件可供选择，还有更好的创新数据库产品值得期待和创造。



Chapter 9 第 9 章

PHP 命令行界面

PHP 不仅可以在 Web 环境下运行，也可以脱离浏览器在命令行和系统后台环境下很好地完成任务。

在本章中，我们将一起了解 PHP 命令行界面（short for Command Line Interface，CLI）的开发，讨论如何在命令行工具下执行，以及如何在单独的服务器进程下运行 PHP 脚本。

PHP CLI 可以在用 Windows 系统的命令提示符运行，也可以在 Liunx/Unix 或者 Mac OS 的终端环境下运行。PHP CLI 有些类似于 CGI 模式，它们之间有很多共同的行为，但 CLI 和 CGI 分属于不同的 SAPI。我们可以理解 CLI 是一个精简的 CGI，没有 GET 或 POST 处理，无 MIME 的头信息输出，以及其他隐含的 SAPI 的版本。

9.1 CLI 简述

9.1.1 CLI 的测试安装

下面为了顺利测试 CLI 模式，我们需要验证有可用的 PHP CLI 界面。在 Windows 系统下，当 PHP 软件包解压后，最好为你的 PHP 所在的根目录（php.exe 所在的文件夹）设置 PATH，这样就可以从命令行中直接执行 PHP。

9.1.2　CLI 的配置参数

我们先了解有关 CLI SAPI 运行时对 php.ini 配置文件进行强制覆盖的参数，以适合运行在 Shell 或命令行环境。

（1）CLI 默认以安静模式开始，不会输出任何 HTTP 头信息。

（2）在运行时，不会把工作目录改为脚本的当前目录（可以使用 -C 和 -no-chdir 参数来兼容 CGI 模式）。

（3）出错时输出纯文本的错误信息（非 HTML 格式）。

（4）CLI SAPI 强制覆盖了 php.ini 中一些参数的设置：

- html_errors：CLI 默认为 False，不显示 HTML 格式错误。
- implicit_flush：CLI 默认为 True。
- max_execution_time：CLI 默认为 0（即没有超时限制）。
- register_argc_argv：CLI 默认为 True，即通过 argc 传递参数。

关于 implicit_flush 和 register_argc_argv 介绍如下：

```
implicit_flush = On
```

强制 flush（刷新），让 PHP 告诉输出层在每个输出块之后自动刷新自身数据，等效于在每个 print() 或 echo() 调用和每个 HTML 块后调用 flush() 函数。

```
register_argc_argv = On ;
```

这条命令告诉 PHP 是否声明 argv 和 argc 变量。变量 $argc 提供传递给应用程序的参数数量，数组 $argv 里保存着实际的参数值。

- 在 PHP CLI 下，有 3 个常量定义在 Shell 环境下，分别为 STDIN、STDOUT 和 STDERR，包括了在 Shell 设备下文件处理的所有情况，如 STDIN（标准输入）：fopen('php://stdin', 'r')。因此，我们可以从 STDIN 读到一行数据，类似于“$strLine = trim(fgets(STDIN));”。

STDIN 已经定义在 PHP CLI 下。在 Windows 下，我们会看到 PHP 的目录，负责解释 CGI 版本的文件为 php-cgi.exe，负责给 CLI 解释的文件为 php.exe。

CLI 模式对于开发如后端静默运行数据等功能非常有用，可以不需要打开浏览器就运行某个 PHP 去抓取，也不会存在超时的情况。CLI 模式在程序运行完之后，工作台界面会立即关闭，并且占用系统资源要比 CGI 方式小得多。

除了上述特征外，CLI 脚本与 Web 下的 PHP 脚本没有什么不同，也需要用 <?php ?> 来包含代码。

9.2 CLI 命令行接口

如果开发环境是 Linux/Unix 系统，要运行 CLI，则需要在 PHP 编译时加入参数。如果在 FreeBSD 环境下编译 PHP，使用 make config 时需要选择 CLI 选项，并且将路径包含在系统的环境变量中，默认的路径是 /usr/local/bin 或 /usr/bin。

在 Lnux 系统下，一个简单的 PHP CLI 脚本的内容类似于下面的内容：

```
#!/usr/local/bin/php -q
<?php
  echo "我是PHP CLI输出的内容!";
?>
```

Windows 用户需要在第一行指定 php.exe 的位置，即可运行，如下面例子：

```
#!C:\php\php.exe -q
<?php
  echo "我是PHP CLI打印的内容!";
?>
```

在 UNIX/Linux 的 Shell 环境下执行时，需要把该文件置为可运行属性：

```
chmod u+rx phpCli.php //或使用755
```

执行该文件：

```
php Cli.php
```

在 Windows 的命令行下可以直接执行 PHP。但事先声明一下 PATH 路径，或者加到系统环境变量中。

9.3 CLI 命令选项

PHP 命令行界面应用起来非常容易，同样也非常稳定，它在命令行下提供很多灵活的选项。最简单的方法就是在命令行下输入（前提：需要设置 PATH 路径指向 php-win.exe 的位置），如下脚本实例：

```
php d:\cli\run.php
```

下面是关于 CLI 命令的语法格式：

- php [options] [-f] <file> [--] [args...]
- php [options] -r <code> [--] [args...]
- php [options] [-B <begin_code>] -R <code> [-E <end_code>] [--] [args...]
- php [options] [-B <begin_code>] -F <file> [-E <end_code>] [--] [args...]
- php [options] [--] [args...]
- php [options] -a

有关 [option] 选项可以使用下面的参数。

-a	交互式运行 PHPCLI 程序
-c <path>|<file>	告诉 PHP，从哪个路径寻找 php.ini
-n	告诉 PHP，不要使用 php.ini 文件
-d action[=input]	定义 php.ini 中的某个参数 action 的值为 'input'
-e	对调试器（debugger）/ 解析器（profiler）生成扩展信息
-f <file>	解析文件 <file>
-h	显示帮助信息
-i	PHP 信息
-l	很有用的选项，进行 PHP 语法检查，不执行
-m	显示在模块编译
-r <code>	运行 PHP <code> 用脚本标识 <?..?>
-B <begin_code>	运行 PHP <begin_code> 在输入一行前
-R <code>	运行 PHP <code> 对输入的每一行
-F <file>	解析和执行 <file> 对输入的每一行
-E <end_code>	运行 PHP <end_code> 在所有行输入后操作
-H	对外部工具隐含每个传递的参数
-s	对源代码语法显示高亮
-v	显示版本号
-w	源代码显示，不显示备注与空白字符
-z <file>	从 <file> 调入 Zend 扩展

PHP CLI 参数（args）通过标准输入（stdin）传递参数。

在 Windows 系统下，我们可以在命令行下显示 PHP CLI 的版本。

```
C:/wamp/php/php -v
```

因为我的机器装了几个版本的 PHP，可能和你的目录有所差别，你在测试时需要改成你的路径。界面如图 9-1 所示。

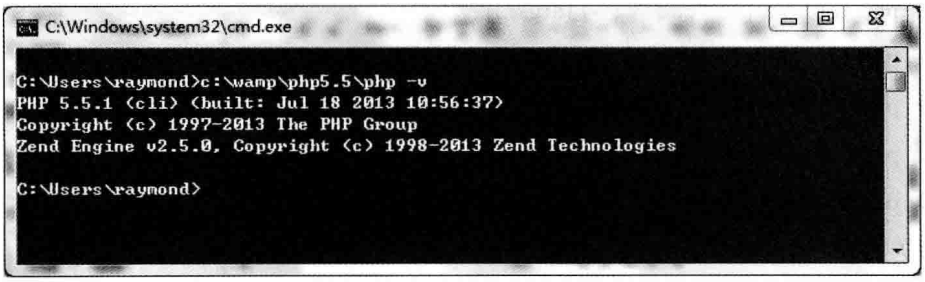


图 9-1　显示界面

仍然在此窗口中，输入 php –i，如图 9-2 所示。

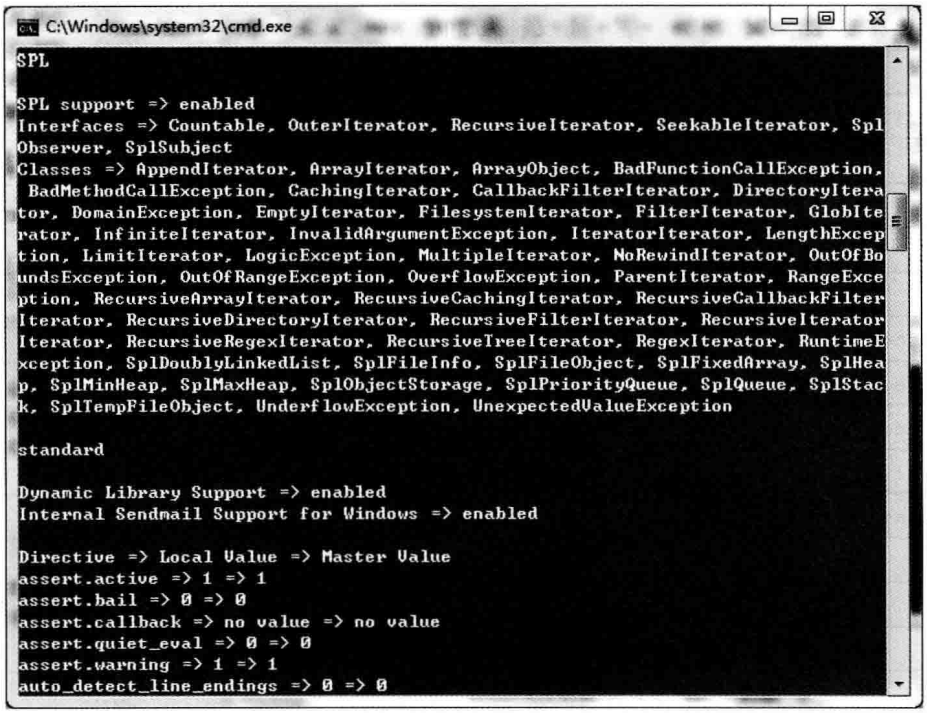


图 9-2　输入 php-i

你可以看到，这和在浏览器中显示 phpinfo() 函数相当。这个窗口的信息量很大，需要不断下拉才能看全所有信息。

我们接下来使用如下命令，查看当前 PHP 环境安装有多少模块，如图 9-3 所示。

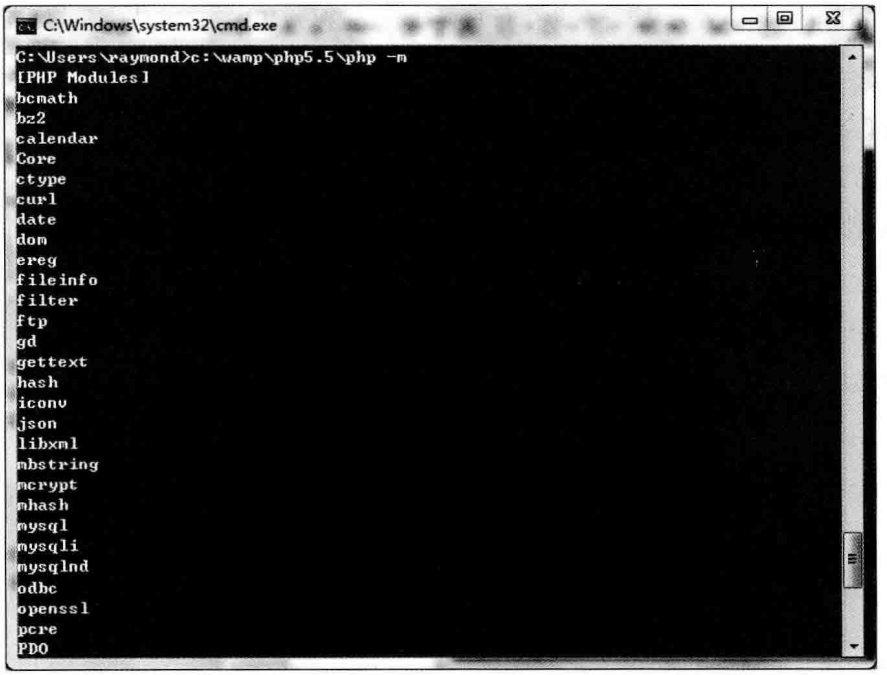


图 9-3　查看窗口

该命令列出了当前环境支持了哪些扩展。这相当重要，比如我们想在 CLI 界面下与 MySQL 数据库交互。

另外还可以使用 -h 参数，呼出 PHP 命令行的更多参数。另外，我的窗口没有特别的设置，为了显示得更好看，你可以修改窗口属性为自己喜欢的前景、背景色。

9.4　CLI 开发实践

PHP CLI 模式开发不需要任何一种 Web 服务器（包括 Apache 或 MS IIS 等），这样，CLI 可以运行在各种场合。

有两种方法可以运行 PHP CLI 脚本。

第一种方法是使用 php -f /path/to/yourfile.php。调用 PHP CLI 解释器，并给脚本传递参数。这种方法首先要设置 PHP 解释器的路径，Windows 平台在运行 CLI 之前，需设置类似 path c:\php 的命令，也失去了 CLI 脚本第一行的意义，因此不建议使用该方法。

第二种方法是首先运行 chmod+x <要运行的脚本文件名 >（UNIX/Linux 环境），将该 PHP 文件置为可执行权限，然后在 CLI 脚本头部第一行加入声明（类似于 #!/usr/bin/php

或 PHP CLI 解释器位置），接着在命令行直接执行。这是 CLI 首选方法，建议采用。

1. 执行一段代码

PHP 命令行界面的基本作用之一就是对一段代码进行测试，并且不需要通过编写脚本、上传到 Web 服务器目录中、使用浏览器测试这一系列步骤。

我们使用 PHP 命令行直接测试或运行小段的 PHP 代码。根据上面的命令行帮助，我们使用：

```
Php -r ‘php code hrere’
```

我们以一个小段代码来进行测试：

```
Php -r ‘echo “hello cli”’ ;’
```

使用这个语法时，需要注意单双引号的分界，另外里面的代码也要正常以“;”分界。

以下是在我的阿里云主机 CentOS Linux 上的执行结果，如图 9-4 所示。

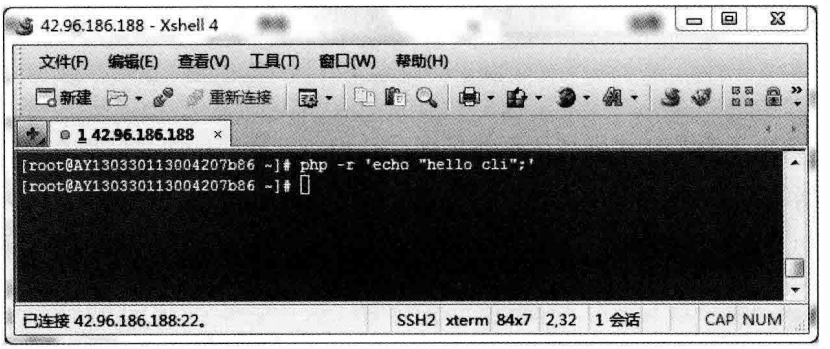


图 9-4 执行结果

但在我的 Windows 7 系统下的同样代码会有解析错误，如图 9-5 所示。

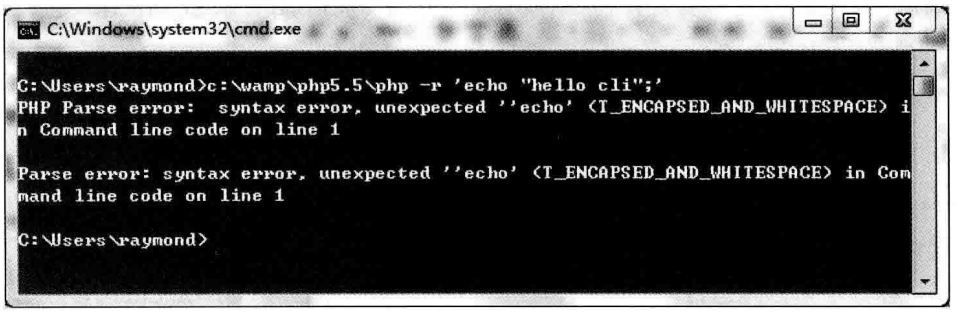


图 9-5 错误解析

我们将单引号换成双引号后正常，如图 9-6 所示。从这点上来看，Windows 和 Linux 系统在解析上有所区别，请各位注意。

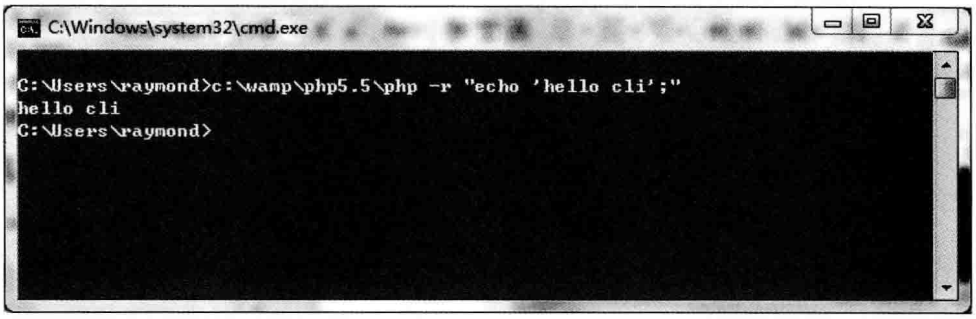


图 9-6　正常页面

2. 可交互的 CLI 界面

在 PHP5.X 以后，PHP 编译后会默认加上 readline 扩展，这样我们就可以使用交互式的命令来运行 PHP。这个交互式 Shell 可以允许持续运行 PHP 命令行界面。命令行如图 9-7 所示。

```
php -a
```

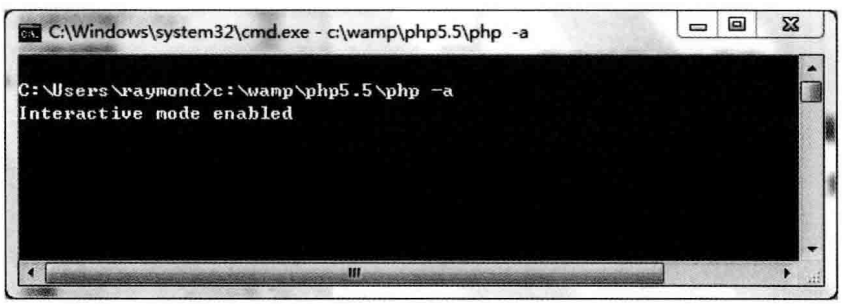


图 9-7　命令行

显示这个结果表示 readline 扩展没有打开，表示交互式 shell 不能使用。如图 9-8 所示。

我们注意到，PHP 会执行这些输入的代码，之后再次显示提示符，用于我们继续输出代码。这个提示符也会友好地显示在下一行，而不像上面的合并在一行。

和正常开发 PHP 一样，我们可以方便地创建一个变量：

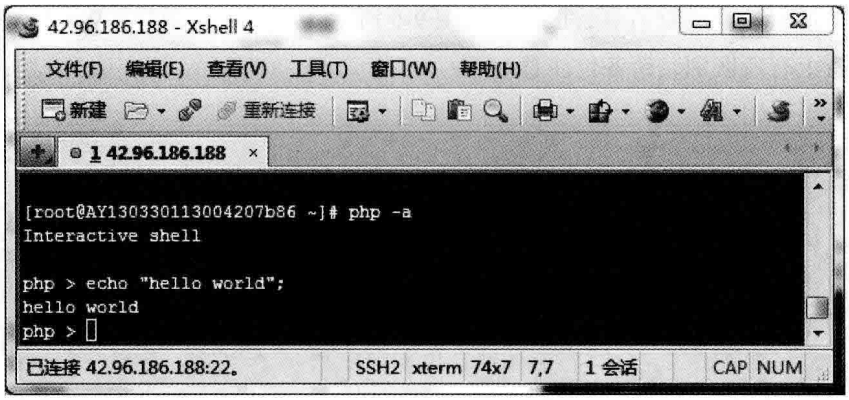


图 9-8 显示结果

```
$arr = array( 'who' ,' what' ,' when' );
Echo $arr;
```

你看，和正常写代码一样，我们可以创建变量、常量、函数并在一个会话周期中引用它们。我们可以使用 TAB 来激活自动完成 / 补全功能。如图 9-9 所示。

图 9-9 使用 TAB

输入 exit，可以正常退出 CLI 交互会话。下面我们来看看怎样创建 PHP CLI 脚本。

3. 创建第一个 CLI 脚本

首先创建一个名为 myfile.php 的 PHP 脚本，用于运行 PHP CLI。该脚本很简单，仅显示“你好 PHP CLI!”。该脚本代码如下：

```
#!/usr/local/bin/php -q
<?php
//Windows平台上，上行应该为：#!C:\php\php.exe -q
echo "你好 PHP CLI!";
?>
```

不要忘了给该文件设置为可执行的权限：

```
$ chmod 755 myfile.php
```

然后直接输入以下命令，按回车键即可以运行：

```
$ ./myfile.php
```

如果要在 Windows 系统下运行该脚本，则不需要设置文件属性，就可以直接运行该脚本。

```
Microsoft Windows [版本 6.0.6000]
版权所有 (C) 2006 Microsoft Corporation。保留所有权利。

C:\ >php myfile.php
你好 PHP CLI!
```

如果在 Windows 平台，CLI 脚本的第一行一定要写正确 php.exe 所在的位置，像这样（另外，如果要在 CLI 脚本中加注释语句，则要把注释写在 PHP 标签里面，因为 CLI 解释只认识第一行，不在 PHP 标签里被认为是语法错误）：

```
#!C:\php\php.exe -q
```

这样，可以看到在命令行下信息已经打印出来，证明该 CLI 脚本已经成功运行。

4. 从命令行上读取参数

如果想从命令行获取参数，CLI 可以从 $_SERVER['argc'] 和 $_SERVER['argv"] 取得参数的个数和值。我们再建立一个文件，名字为 testargs.php，脚本代码如代码清单 9-1 所示：

代码清单9-1　testargs.php

```
#!C:\php\php.exe -q
<?php
```

```
//UNIX和Linux平台下应该为#!/usr/local/bin/php -q
echo "测试获取参数:\n";
echo $_SERVER["argc"]."\n";
//显示传入的参数值，从索引1开始显示
echo $_SERVER["argv"][1]."\n";
echo $_SERVER["argv"][2]."\n";
echo $_SERVER["argv"][3]."\n";
echo $_SERVER["argv"][4]."\n";
?>
```

在命令行输入如下代码：

```
C:\Users\John>testargs.php Always To Be Best
```

测试获取参数：

```
4
Always
To
Be
Best
```

因为我们输入了一串单词，为“Always To Be Best”，脚本参数以空格分隔。因此，PHP 将其计为 4 个参数，下面对此说明。

$_SERVER["argc"] 数组返回一个整型的数，代表从命令行上回车后一共输入了几个参数。

从上例的结果已经看出，要访问已经传入的参数值，需要从索引 1 开始。因为脚本本身的文件已经占用了索引 0，即 $_SERVER["argv"][0]。

5. 处理 I/O 通道

PHP 设计最初并不是为了与用户直接的键盘输入或文本输出结合使用。了解这一设计至关重要，因为如果需要在命令行中执行任何操作，都必须能够与用户来回通信。

输入输出（I/O）通道这个思想来源于 UNIX 系统，UNIX 系统提供 3 个文件句柄，用以从一个应用程序及用户终端发送和接收数据。

我们可以把一个脚本的输出重定向到一个文件：

```
php world.php > outputfile
```

如果是在 UNIX 系统下，也可以使用通道定向到另一个命令或应用程序中。例如：

```
php world.php | sort.
```

在 PHP 5 的 CLI 中，有一个文件流句柄，可以使用 3 个系统常量，分别为 STDIN、

STDOUT 和 STDERR。下面我们分别介绍。

（1）STDIN。

STDIN 全称为 standard in 或 standard input，标准输入可以从终端取得任何数据。

格式：stdin ('php://stdin')

下面的例子是显示用户输入：

```
#!/usr/local/bin/php -q
<?php
    $file = file_get_contents("php://stdin", "r");
    echo $file;
?>
```

这段代码的工作原理与 cat 命令很相似，回转提供给它的所有输入。但是，这时它还不能接收参数。

STDIN 是 PHP 的标准输入设备，利用它，CLI PHP 脚本可以做更多的事情。如代码清单 9-2 所示：

代码清单9-2　STDIN CLI脚本

```
#!/usr/local/bin/php -q
<?php
//UNIX平台下第一行应该为#!/usr/bin/php -q
/* 如果STDIN未定义，将新定义一个STDIN输入流 */
if(!defined("STDIN")) {
define("STDIN", fopen('php://stdin','r'))
}
echo "你好!你叫什么名字(请输入):\n";
$strName = fread(STDIN, 100); //从一个新行读入80个字符
echo '欢迎你'.$strName."\n";
?>
```

该脚本执行后将显示：

```
你好!你叫什么名字(请输入):
```

比如，输入 Raymond 之后，将显示：

```
欢迎你Raymond
```

（2）STDOUT。

STDOUT 全称为 standard out 或 standard output，标准输出可以直接输出到屏幕（也可以输出到其他程序，使用 STDIN 取得），如果在 PHP CLI 模式里使用 print 或 echo 语句，则这些数据将发送到 STDOUT。

格式：stdout ('php://stdout')

我们还可以使用 PHP 函数进行数据流输出。如代码清单 9-3 所示：

代码清单9-3 PHP数据流输出

```
#!/usr/local/bin/php -q
<?php
$STDOUT = fopen('php://stdout', 'w');
fwrite($STDOUT,"Hello World");
fclose($STDOUT);
?>
```

输出结果如下：

```
Hello World
```

例如，echo 和 print 命令打印到标准输出。如代码清单 9-4 所示：

代码清单9-4 打印到标准输出

```
#!/usr/local/bin/php -q
Output #1.
<?php
echo "Output #2.";
print "Output #3."
?>
```

这将得到：

```
Output #1.
Output #2.Output #3.
```

PHP 标记外的新行已被输出，但是 echo 命令或 print 命令中没有指示换行。事实上，命令提示符重新出现在 Output #2.Output #3. 所在的行中。PHP 拥有的任何其他打印函数将会像此函数一样运行正常，任何写回文件的函数也是一样的。如代码清单 9-5 所示：

代码清单9-5 使用STDOUT输出通道

```
#!/usr/local/bin/php -q
<?php
    $STDOUT = fopen("php://stdout", "w");
    fwrite($STDOUT, "Output #1.");
    fclose($STDOUT);
?>
```

以上代码将把 php://stdout 作为输出通道显式打开，并且 php://output 通常以与 php://

stdout 相同的方法运行。

（3）STDERR。

STDERR 全称为 standard error，在默认情况下会直接发送至用户终端，当使用 STDIN 文件句柄从其他应用程序没有读取到数据时会生成一个“stdin.stderr”。

格式：stderr ('php://stderr')

下面的脚本表示如何把一行文本输出到错误流中。如代码清单 9-6 所示：

代码清单9-6　输出到错误流

```
#!/usr/local/bin/php -q
<?php
$STDERR = fopen('php://stderr', 'w');
fwrite($STDERR,"There was an Error");
fclose($STDERR);
?>
```

PHP 5.2 可以直接使用 STDOUT 作为常量，而不是定义上面使用的变量 $STDOUT，为了兼容之前版本，我们仍使用了自定义变量，如果你使用的是 PHP 5.2，则可以参考 STDIN 的第二个例子。

6. 后台运行 CLI

如果正在运行一个进程，而且在退出账户时该进程还不会结束，即在系统后台或背景下运行，那么就可以使用 nohup 命令。该命令可以在退出账户之后继续运行相应的进程。

nohup 在中文中就是不挂起的意思（no hang up）。该命令的一般形式为：

```
nohup -f scriptname.php &
```

使用 nohup 命令提交作业，在默认情况下该作业的所有输出都被重定向到一个名为 nohup.out 的文件中，除非另外指定了输出文件。

```
nohup scriptname.php > log.txt &
```

这样，PHP CLI 脚本执行后的结果将输出到 log.txt 中，我们可以使用 tail 命令查看该内容：

```
tail -n50 -f log.txt
```

现在再来实现两个例子，第一个是每隔 10 分钟自动生成一个静态 HTML 文件，并一直执行下去。脚本代码如代码清单 9-7 所示：

代码清单9-7 定期生成静态页面一例

```
#! /usr/local/bin/php
<?php
set_time_limit(0);
while(true){
@fopen("/usr/local/www/data-dist/content/article_".time().".html","w");
sleep(600);
}
?>
```

保存并且退出 vi 编辑器，然后赋予 genHTML.php 文件可执行权限：

```
#>chmod 755 genHTML.php
```

然后让脚本在后台执行，执行如下命令：

```
$nohup genHTML.php -f &
```

执行上述命令后出现如下提示：

```
[1] 16623
```

按回车键后将出现 shell 提示符。上面的提示就是说，所有命令执行的输出信息都会放到 nohup.out 文件中。

执行上面命令后，每隔 10 分钟就会在指定的目录生成指定的 HTML 文件，如 article_111990120.html 等文件。

如何终止 CLI 程序的后台运行呢?

可以使用 kill 命令来终止这个进程，终止进程之前要知道进程的 PID 号，即进程 ID，我们使用 ps 命令：

```
www# ps
  PID  TT  STAT      TIME COMMAND
  561  v0  Is+    0:00.00 /usr/libexec/getty Pc ttyv0
  562  v1  Is+    0:00.00 /usr/libexec/getty Pc ttyv1
  563  v2  Is+    0:00.00 /usr/libexec/getty Pc ttyv2
  564  v3  Is+    0:00.00 /usr/libexec/getty Pc ttyv3
  565  v4  Is+    0:00.00 /usr/libexec/getty Pc ttyv4
  566  v5  Is+    0:00.00 /usr/libexec/getty Pc ttyv5
  567  v6  Is+    0:00.00 /usr/libexec/getty Pc ttyv6
  568  v7  Is+    0:00.00 /usr/libexec/getty Pc ttyv7
16180  p0  I      0:00.01 su
16181  p0  S      0:00.06 _su (csh)
16695  p0  R+     0:00.00 ps
16623  p0  S      0:00.06 /usr/local/bin/php /usr/local/www/data/genHTML.php
```

已经看到 PHP 的进程 ID 是：16623，于是再执行 kill 命令：

```
$ kill -9 16623
[1]+  Killed                  nohup /usr/local/www/data/genHTML.php
```

这时该命令的进程就已经被终止了，再使用 ps 命令：

```
$ ps
PID  TT  STAT      TIME COMMAND
82374  p3  Ss     0:00.17 -bash (bash)
82535  p3  R+     0:00.00 ps
```

刚才的 PHP CLI 脚本已经没有了，如果直接运行 ps 命令无法看到进程，那么就结合使用 ps & apos 两个命令来查看。

上面例子必须运行在 UNIX 或 Linux 系统中，在 Windows 环境不支持 nohup。

9.5　CLI 实际应用

以下是一个使用 PHP 编写对 MySQL 数据库备份的 PHP CLI 脚本程序（仅适用于 UNIX 或 Linux 系统，当然也可以改写和优化），如代码清单 9-8 所示：

代码清单9-8　使用CLI对MySQL数据库备份

```
#!/usr/local/bin/php -q
<?php
    //配置数据库信息
    $dbhost = 'localhost';
    $dbuser = 'username';
    $dbpass = 'password';
    $mysqldump = '/usr/local/mysql/bin/mysqldump --opt --quote-names';
    //接受参数，如果参数不足，提示
    if ( count( $argv ) < 2 ) {
      ?>
      backupDatabase.php
      Create a backup of one or more MySQL databases.

      Usage: <?=$argv[0]?> [$database] $path

      $database -
        Optional - if omitted,  default is to backup all databases.
        If specified,  name of the database to back up.
```

```
  $path -
    The path and filename to use for the backup.
    Example: /var/dump/mysql-backup.sql
  <?php
  exit();
}

//如果数据库的参数遗漏
$database = NULL;
$path = NULL;
if ( count( $argv ) == 2 ) {
  $database = '--all-databases';
  $path = $argv[1];
}
else {
  $database = $argv[1];
  $path = $argv[2];
}

//构造命令
$command = "$mysqldump -h $dbhost -u $dbuser -p $dbpass $database > $path";

// create a version of the command without password for display
$displayCommand = "$mysqldump -h $dbhost -u $dbuser -p $database > $path";
print $displayCommand . '\n';

//在shell运行该命令并且验证备份是否成功
$result = shell_exec( $command );
$verify = filesize( $path );
if ( $verify ) {
  print "\nBackup complete ($verify bytes).\n";
}
else {
  print '\nBackup failed!!!\n';
}
?>
```

有时候，我们需要批量处理目录属性，以下的 PHP Shell 脚本（只适用于 UNIX 和 Linux 系统）会比 chmod 工具更方便，如代码清单 9-9 所示：

代码清单9-9　批理处理目录权限与属性

```
#!/usr/local/bin/php -q
<?php
    //预设置目录属性（样例）
```

```
$presets = array( 'production-www'=>'root:www-0750',
                  'shared-dev'=>':www-2770',
                  'all-mine'=>'-0700'
                  );

//稍候再发送警告信息
ob_start();
?>
resetPermissions.php
Changes file ownership and permissions in some location according
to a preset scheme.

Usage: <?=$argv[0]?> $location $preset

  $location -
    Path or filename. Shell wildcards allowed

  $preset -
    Ownership / group / permissions scheme,  one of the following:
    <?php
       foreach( $presets AS $name=>$scheme ) {
         print $name . '<br />';
       }
$usage = ob_get_contents();
ob_end_clean();
// provide usage reminder if script was invoked incorrectly
if ( count($argv) < 2 ) {
  exit( $usage );
}
// 导入参数
$location = $argv[1];
$preset = $argv[2];
if ( !array_key_exists( $preset,  $presets ) ) {
  print 'Invalid preset.\n\n';
  exit( $usage );
}
// 解析参数 [[$owner]:$group][-$octalMod]
// 第一个参数导入到变量$properties中
$properties = explode( '-',  $presets[$preset] );
// determine whether chown or chgrp was requested
$ownership = FALSE;
$owner = FALSE;
$group = FALSE;
if ( !empty($properties[0]) ) {
  $ownership = explode( ':',  $properties[0] );
```

```
        if ( count( $ownership ) > 0 ) {
          $owner = $ownership[0];
          $group = $ownership[1];
        }
        else {
          $group = $ownership[0];
        }
      }
      // 检查chmod操作是否已经成功
      $octalMod = FALSE;
      if ( !empty( $properties[1] ) ) {
        $octalMod = $properties[1];
      }
      //执行相应的系统命令
      $result = NULL;
      if ( $owner ) {
        print "Changing ownership to $owner.\n";
        $result .= shell_exec( "chown -R $owner $location 2>&1" );
      }
      if ( $group ) {
        print "Changing groupership to $group.\n";
        $result .= shell_exec( "chgrp -R $group $location 2>&1" );
      }
      if ( $octalMod ) {
        print "Changing permissions to $octalMod.\n";
        $result .= shell_exec( "chmod -R $octalMod $location 2>&1" );
      }
      // 显示错误信息
      if ( !empty( $result ) ) {
        print "\nOperation complete,  with errors:\n$result\n";
      }else {
        print 'Done.\n';
      }
?>
```

一个实用的文件压缩脚本，实际上是调用tar工具来进行压缩，如代码清单9-10所示：

代码清单9-10　使用CLI调用压缩工具

```
#!/usr/bin/php -q
<?php
    /* This first if is there to make sure that there are exactly two arguments.
     * The first argument would be "backup.php" because thats the name of the
        script,
     * The second argument would be the directory name that they want to back up
```

```
 */

if($_SERVER['argc'] != 2) {
    die('backup.php Expects Only one argument: Usage: backup.php <directory-
        name>')
}

/* This tests to see if the file really exists using file_exists()
 * And that its a directory using is_dir(), if the file doesn't exist
 * or isn't a directory, it would die with an error
 */
if(!file_exists($_SERVER['argv'][1]) OR !is_dir($_SERVER['argv'][1])) {
    die($_SERVER['argv'][1] . ' is not a valid directory');
}

/* This will execute the linux command that should tar the the directory,
 * Tar is a program a lot like zip that should shove the contents of the
     directory
 * into a "backup.tar" file.
 */
exec('tar -cvf backup.tar ' . $_SERVER['argv'][1]) or die('Tar failed!');
echo "File Backed up to backup.tar Sucessfully!";
?>
```

9.6　内置服务器

PHP5.4 为开发者提供了一个内置的 Web 服务器，它的作用就是方便我们做测试开发使用，不能指望它代替 Apache 或 Nginx。

确认你的 PHP 版本是 5.4 及以上后，我们输出如下命令来确认 Web 服务器是否可用：

```
Php -h
```

我的服务器使用的 PHP 5.6，其显示文档中包括 -S 和 -t 选项，表示支持内置服务器。如图 9-10 所示。

切换到包括我们想要测试的目录：

```
cd /var/www/dev.21cto.com/clitest
```

启动测试 Web 服务器：

```
php -S localhost:8000
```

如果你想更完美点，还可以加上 IP 地址，如下：

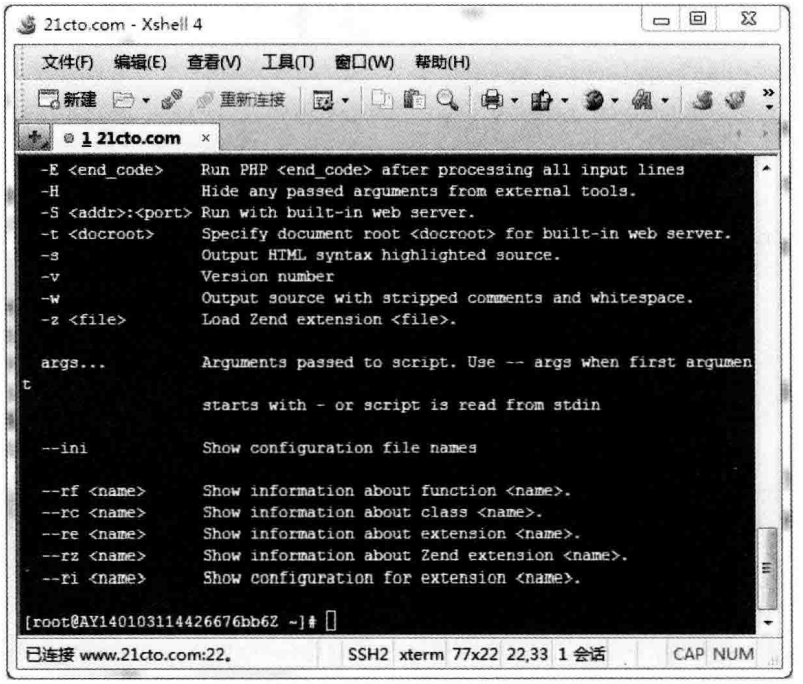


图 9-10 输出页面

```
Php -S 115.28.174.91:8000
```

使用浏览器浏览 http://dev.21cto.com:8080/script.php（主机名请更换为正式环境）即可。我们可以刚才的目录创建一个文件叫作 phpinfo.php，里面显示 phpinfo() 函数。如图 9-11 所示。

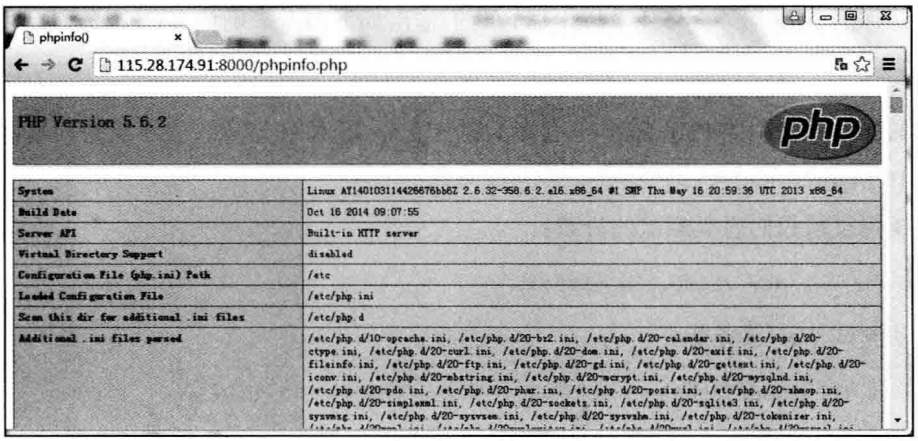


图 9-11 phpinfo() 函数

我们可以在 CLI 控制台上看到实时的请求记录，如图 9-12 所示。按 Ctrl+C 可以退出该窗口。

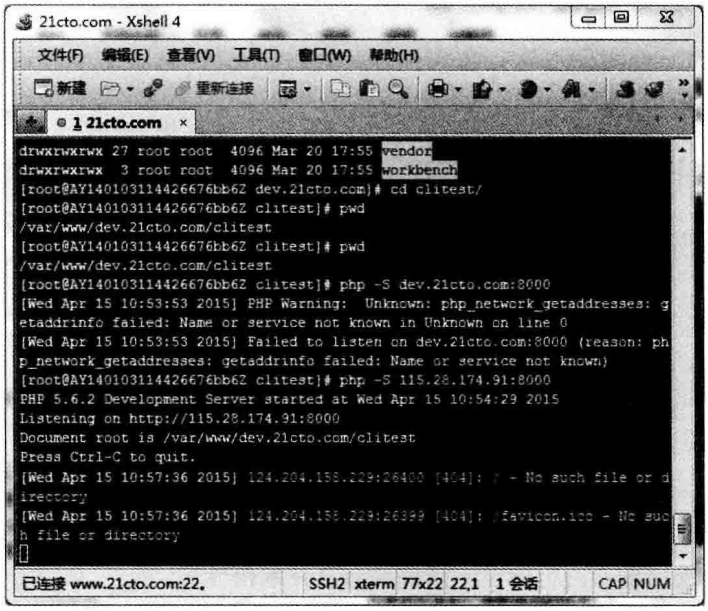


图 9-12　请求记录

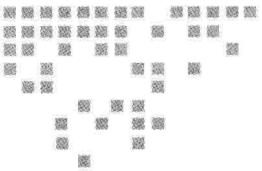
上图中可以看到第一条服务器请求记录 Document root 告诉我们是从哪里启动的。

9.7　本章小结

本章着重讲述了 PHP 在工作台下运行的命令行界面模式。我们一起探讨了如何启动一个 PHP CLI 模式和实用的操作方式。

PHP 的这种模式非常有用，使用它可以作为后端处理的应用，如果与 UNIX 的 crontab 或 Windows 端的计划任务配合使用，则可以更大地发挥 PHP 的优势。

其实我非常喜欢用 PHP CLI 命令行，有种回到 DOS 时代的感觉，如果你也喜欢，请你多练习吧！



Chapter 10 第 10 章

代码重构实践

开发者要成长得更优秀，真正提升自己的能力和水平，首先需要先从思想开始转变，踏实勤勉、精益求精地做践行者，不断提高自身技术悟性。

试着回想曾经的开发经历，有写得好也有不好的，有时候我们肯定会想，什么是好代码，怎么样让自己的代码写得更好呢？

本章的主题就是围绕如何写好代码来描述，重点介绍编码艺术与代码重构策略，一起来讨论为什么、什么时候、怎么样、在什么地方对代码进行重构。

代码重构是一门实践性极强的艺术，并不是“高大上”、要束之高阁的科学，我们要了解更多隐藏在代码后面的内容，并非写完程序就万事大吉，那只是懒惰的程序员所为，优秀的开发者要承担代码的所有责任。

正如我们工作时遇到的问题，有时候就是代码带来的副作用，亟待我们帮它清理、瘦身，让它们变成清新代码的同时，让它运行的性能更佳。

10.1 什么是不良代码

在一些情况下，身担上线重任的我们要完成开发任务，代码逻辑没有仔细考虑，常常会出现代码冗余，有的甚至是从其他文件中复制过来的。如果复制的代码也存在错误，这样会让代码越来越混乱，导致错误更大的蔓延。这些“坏味道”的代码主要表现在如下方面：

- 代码逻辑难以被读懂理解。
- 代码中要增加很多单行注释。
- 代码逻辑存在低级的 Bug。
- 存在冗余的代码。
- 过多面向过程的代码。

我们开发的网站以及系统架构，对于所在公司来说是重要的智慧财产，如果代码过多冗余，对于公司的运营和后续开发调试都会增加困难。

10.2　什么是好代码

我们的中医讲究“望、闻、问、切”，鉴定代码的好坏亦可沿用此道理。你如果写过一些程序了，会对此有所感触，会比较容易地鉴别代码的好与坏。如果你刚刚从事开发，想从开始就能做正确的事，该怎样去做?

我们通过 3 个准则来判断代码是不是好代码，分别是：

- 可读性。
- 可扩展性。
- 效率。

下面，我们就这 3 个准则来分别详细说明。

1. 可读性

通俗来讲，我们在项目中按照一些标准的语法、风格来开发软件，如果在解释代码逻辑时很容易讲清楚、讲明白，初步判断就是一篇可读性较好的代码。

我们可以遵循一个通用标准，或者自己定义一个风格。无论你选择哪种，当同事接替你的工作，按照你的代码、风格以及文档、注释等就能够顺利工作，而无须太多解释，这也是干净的好代码。

在讲究严谨的研发团队中，成员都遵守通用的编码标准，当有新人加入团队，可以有效缩短学习曲线和有效减少开发成本。

特别是产品第一版本完成开始迭代时，同事可能会接续你的工作，可读性和可扩展性的重要性就开始体现，比如你使用的语言的标准，为解决问题使用了哪些设计模式。

如果你开发时遇到新的难题，设计了很好的解决方案，非常有必要在代码里说明，

包括如何使用以及实现的逻辑。如果有注释，其他的程序员就可以了解你提供的是很有价值的方案，也更能了解你的思维，可能会说："这家伙是特别能写程序的天才！"

2. 可扩展性

可读性与可扩展性有助于代码的可维护性，使得在项目规划的时间内能快速迭代。

很明显，可读性好的代码维护性高，此外，代码逻辑也需要可维护性，也就是使用设计模式，它有助于可读性和可扩展性。

软件开发中遵循一些可重复使用的、逻辑的、已知的设计模式，它是定义的标准设计模式或正常的逻辑流程，因此代码具备更大的可扩展性。

如一些开发者喜欢用单件模式（Singleton）去连接 MySQL。如下代码：

```
<?php
class db{
    private static $instance = false;
    public static function instance($host,$name,$pass,$dbname){
        if(self::$instance===false){
            self::$instance=new db(......);
        }
        return self::$instance;
    }

    private function __construct(){
        /*这个函数须是私有的，它保证了只能通过单例生成器来获得实例*/
    }

    public function __clone(){
        /*什么也不做*/
    }
}
```

现在我们无论在哪个文件里，都可以使用：

```
$db = db::instance(......);
```

另外，在遍历数组时通常我们会用 foreach 语句，这已经是 PHP 编程的标准方法。

但有的人会喜欢使用 while 或 for 这样的循环语句，除非有特殊情况，我们实在不该使用这样的结构，避免不必要的混乱，造成可维护性不佳。

可扩展性的主要体现是去除耦合和封装。去除耦合意味着代码（主要是函数 / 方法和类）不相互依赖或相互重叠，而应该是"纯粹"的功能实现，任何重叠的功能和其他非关联性实体都应该删除。

我们正在写的函数逻辑应该与它的命名相符，不能“挂羊头卖狗肉”，即不在函数内做额外的东西。

在编写代码时使用“工具箱”的方式——众多函数库共同打造大的复杂系统。

封装也是去除耦合的重要部分。要封装组件，就要从项目的范围抽取逻辑，并分离其内部相互作用缩小在接口级别，这是一个通用模块化的方法，这样使得它更容易在以后删除或更新系统的任何功能，使代码更清晰。

3. 效率

在开发过程中，我们应在头脑中时刻牢记“效率”二字，效率说开来即系统的“瓶颈”问题。

因为在任何项目的代码中都有使用一些不合理的结构：例如嵌套循环和递归。还有一些不正确的处理逻辑，例如不使用缓存和发送 / 接收数据时没有适当的使用缓存。

互联网产品中，可读性和可扩展性不好导致的结果就是运行效率降低，大多数情况下，效率与这两个规则直接成正比。

如果特殊情况下确实需要使用递归，最好确保风险的规避，以及清晰的头脑，让自己的代码保持清晰干净。

10.3　如何增加代码可读性

在接下来的部分里，重点介绍如何让代码更具可读性。为了帮助大家在重构代码时具备可读性，我准备了一些设问句，你在审查整个编码开发过程中可作为参考。

1. 代码文件是否遵循相同编码风格

你应该使用的编码标准，最好是一个通用的标准，如 PSR-1/2。

如果定义编码标准，你可以以自己的代码为范本，为你自己和你的团队写一个简短的说明是可以的。

2. 开发使用的是标准语法结构，不使用怪语法

由于 PHP 的灵活性，我们可能有多个样式来编写程序结构（如控制语句的不同写法）。如果你使用的是比较特殊的语法，例如使用替代语法，只符合 HTML 模板和 C 风格的语法标准的代码，在同一个项目中需坚持一种风格。

比如使用 C 风格的语法，在项目的文件里，要避免不同的语法混在一起。

3. 每个文件有一个标题，注释、记录其在项目中的角色

作为一项基本的要求，你应该在每个程序文件中包括一个头与下面的注释：

- 应用程序的名称、版本和简要说明。
- 文件到其他文件和包 / 组的从属关系。

如下代码：

```
/**
 * 与淘宝分销平台的商品同步
 * ==========================================================================
 * 版权所有 2008-2014 思技创想（北京）科技有限公司，并保留所有权利。
 * 网址：http://www.21cto.com
 * --------------------------------------------------------------------------
 * 说明：
 * 淘宝商品同步Controller
 * 放在crontab中运行 *
  ==========================================================================
 * @Author: Chris Zhang(chris@21cto.com)
 * @copyright  2007-2014 Think Creative Technology Co.,Ltd
 * @version  Release: $Revision: 17342 $
 * @license  Commerce
 * @Id: auto_product_sync_to_taobao.php 17342 2014-12-08 12:20:30 Chris $
*/
```

4. 通用的代码保持在最低限度

在项目中为加快开发速度和引用方便，我们会使用公共脚本，然后使用 include 包含语句引入当前 PHP 脚本中。

如果一个复杂的 Web 应用程序，使用公用的脚本也会成为一个头痛的问题，如果在文件中使用过多的 include 和 require，有的工程师未必会完全理解，所以要最大限度地减少使用。

如果通用的代码包含其他模块函数和类定义，特别控制结构，需要重构函数或类。作为基本的经验法则，控制结构应该不包括在公共的代码中。

10.3.1 命名方式

1. 类、变量等命名，遵循标准模式和代码风格

在整个代码库中，变量、常量、函数、方法、类、命名空间遵循统一的命名和风格，这是编码标准中的重要规定。

2. 命名有意义且无歧义

PHP 在命名方式上是可圈可点的，我们在编码中使用的变量、函数等名称也需要如此，在命名方面要多考虑，让名字有意义。不要创建类似 $abc 这样的变量名，而像 $count、getFile() 等这样的，命名良好且容易理解的名称看起来就是一个专业人员开发的。

也有例外的情况，比如专业教材或事实上的命名行为，如计数器变量 $i/$j/$ 等用于循环的变量，这些用当然没有任何问题。

3. 命名是否过长或过短

变量名和常量的长度平均在 8～12 个字符，特殊的可以达到 25 个字符，但更重要的是不要命名成为不可读的、烦琐的名字。如果你的代码里有任何非常长（或很短）的命名，那么应该重新考虑它的标识。

有关使用情况的深度思考，有助于想出更好的名称来：

```
public function process($text, $postId)
{
    $mentions = $this->parse($text);
    if (!empty($mentions)) {
      foreach($mentions as $mention) {
    //以下省略
}
```

方法 / 函数的名称应平均控制在 15～25 个字符长度，但有一些例外可能达到 30 个字符。作为变量名，你应该重新考虑长 / 短命名。

而类的名称应尽可能短，并使用名词命名。比如：

```
/**
 * The comment is being viewed. This hook can be used to add additional data
      to the comment before theming.
 *
 * @param $comment
 *   Passes in the comment the action is being performed on.
 * @param $view_mode
 *   View mode, e.g. 'full', 'teaser'...
 * @param $langcode
 *   The language code used for rendering.
 *
 * @see hook_entity_view()
 */
function hook_comment_view($comment, $view_mode, $langcode) {
```

```
    // how old is the comment
    $comment->time_ago = time() - $comment->changed;
}
```

除非我们的项目有特殊的编码标准，要按照其约定来为类命名，类的名称应该控制在 10～15 个字符。

10.3.2 表达式

1. 表达式遵循标准格式与风格

编写表达式与代码格式的可读性是至关重要的。有些人喜欢添加一个空格之前操作数，如 $incounter = 15，而一些人不愿意添加任何空格。无论选择哪一个，所有的代码是一致的即可。

2. 表达式理解复杂还是容易

对于较复杂或太长的表达式，我们应该将它分成多行。

作为一个通用规范，如果表达式包含 3 个或更多操作数，就需要写上注释，用于将来被自己记起或被别人理解。如果更新了表达式，也需要同步更新注释，这点非常重要，请千万不要忘记。

10.3.3 代码段

1. 代码段遵循标准模式 / 风格

代码块是代码的基本组成部分。你在写每一行代码时考虑其风格，因为人的眼睛对代码风格比内容本身更敏感。

保持你的开发标准，比如在合适的地方加括号、空格等，你的眼睛舒服地看到规律的代码结构时，在读代码时更会增进理解。

2. 代码段大于 25 行时需要重构

如果你的一个代码块超过 25 行，我建议至少重构一次，如果你愿意多次重构代码，可以达到最佳效果。

有时代码块会有一些特殊，可能会达到 50 行甚至更多。这种情况还是少数，不是习惯做法。长代码的可读性不够好，越小块代码的可读性和可维护性越大。

3. 有对功能说明的注释

和艺术一样，完美的代码都会说话。我们每个代码块都要有必要的注释，把它当作

一种习惯和规则，就像每天关上门习惯锁门一样。

虽然生活并非尽善尽美，你只需要做到在长于平均水平（超过 20 行）的代码块，或执行复杂的任务或逻辑处加入注释就好。

一般情况下，我会在复杂的逻辑前加入较多的注释文字，而必要的代码块加入单行注释。微不足道的、谁都可以看得懂的语句就不需要注释了。

10.4　可扩展性与效率重构

所谓可扩展性，是可重复使用的、合乎逻辑的、使用已知设计模式的代码。

模块化的代码往往是高度可扩展的，而过程化的代码往往可扩展性低，过程化的代码会更有效率。因此一些常用的做法是，以模块化的方式开发和部署在一个过程代码的方式，可以做到两全其美。

关于可扩展的代码的主要方面是：（正常的逻辑流程和设计模式）逻辑的可扩展性，模块化设计，去除耦合与封装。

下面主要介绍逻辑扩展的内容。

1. 大部分的代码块，按照正常的逻辑流程

我们正在处理的小的逻辑性问题，请使用的是正确的结构（if、foreach、if else 等），我的意思是“结构”的工作，你应该使用最合乎逻辑的语言功能，而有的人可能习惯用三元操作符来代替 if 语句。

例如，通过简单的数组循环用 foreach，这是常见的方式，而在这种简单的迭代使用 for 循环，在 PHP 这样的语言中是不甚合理的。

使用复杂的逻辑来完成一个简单任务，你可能有千般理由，在这种情况下，如果在一段时间后，你能回忆起以前的部分，那证明你是好样的。

2. 复杂问题的解决方案遵循标准设计模式

刚开始使用 PHP 时，那时候我也不了解设计模式，只是实现式地开发。现在发现大型项目中使用设计模式是必须的，因为它们通常容易理解，并能够考虑到将来开发的扩展性。

一般情况下，使用一个标准软件模式来解决一些复杂的问题，要先编写实现的基类。比如什么时候使用 Factory 模式，什么时候使用 Proxy 模式。如果你有兴趣，欢迎阅读本书的设计模式一章。

10.5 模块化设计

1. 代码结构模块化设计

模块化设计的意义就是将产品按业务不同划分为不同的模块。由小的应用程序组合成一个大的应用程序，这样的好处是更容易开发和易于扩展、维护。

其中的每个模块都有自己的特性和功能，可以担当独立的功能和应用。

模块核心功能和应用程序的入口点，我们可以在将来方便地添加新的模块以扩展功能，这就是为什么现在大多数产品都设计为插件模块来进行开发。

因此开发者在应用程序结构设计时，需要有模块 / 插件的安装和卸载功能，需要开发的核心模块，对插件的基本结构进行定义。

一个代码中的一些模块作为单一的实体和比较少的参数，若不是这样就需要把它分解成一个新的模块。当我有一个功能，在一个单独的类中做了越来越多的副作用任务时，就需要毫不犹豫地把这些迁移到一个新的模块。

一个精心设计的应用设计模块化后，可以有效地防止只为一个任务而写的孤儿代码。每当我们写有一些孤儿代码时，我就将它迁移到一个实用模块中，用于处理通用功能和完成小任务，该模块就是通用的函数库，比如 Code Igniter 框架中的 helper 库。

当这些任务足够大、需剥离逻辑，我会再尝试将它们移动到自己的独立的模块中，此种重构是一个持续的过程。

2. 模块依赖性最低

模块自身提供尽可能多的功能，且天然地和其他模块有良好的依赖关系，比如电子商务中“库存”模块依赖于“财务”模块，才能成为完整的电子商务系统。

事实上有许多硬的依赖关系是不好的，它们使调试和部署变得比较困难。为了保证模块间的依赖关系，我们需要遍历当前每一个模块，然后在代码库中看是否有任何模块之间存在硬的依赖关系后清除它们。

如果不能，应该将这两个模块合并到一个模块，并命名为通用的名称。

10.6 封装与解耦

1. 函数、方法和类低耦合，高内聚

比较常见的例子，我们在程序中添加分页功能，从数据库中取得结果显示，这是一种很常见的任务。在我早期的 PHP 开发职业生涯中，会在进行分页的程序里写一些代码

取数据库的结果来分页，构成非常具体的功能。

后来我发现分页很多功能的代码是共通的，也就是能够复制的逻辑或代码，我便把这些共通的代码抽象出来形成一个分页函数或类。诸如此类，你也会发现需要对自己代码解耦，以提高代码的可重用性和可扩展性。

2. 单一职责：模块和组件相对解耦

代码保持最低限度的依赖，这是解耦的正确途径。比如一个类只做一件事，并把这件事做好，且只有一个引起它变化的原因。

单一职责原则可以看作低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而极大地损伤其内聚性和耦合度。单一职责，通常意味着单一的功能，因此不要为一个模块实现过多的功能点，以保证实体只有一个引起它变化的原因。

降低依赖关系重要的指标是可用性和可扩展等复杂度没有增加。

10.7　代码效率

大多数 Web 应用都要实现快速响应，所以效率通常依赖网站开发者，让系统“瓶颈”保持在最低限度，无论是内存消耗，还是 CPU 占用率、可用性和节省网络流量。我们都希望应用程序的速度和 HTML 渲染是“相当快”。

在第 4 章中，我提到一些效率低下的根源，以及应对这些问题的解决方案。我们了解到，当涉及效率时有两个“好朋友”：分别是缓冲（Out Buffer）和缓存（Cache）。内存缓冲与网络资源缓存，可有效地利用高速缓存高效地利用内存和处理器，让 Web 应用具备较高性能。

我们还可以在粒度更细的层面重构代码。举一个例子，比如电子商务网站中生成缩略图图片库，此时应该使用缓存，而不是动态生成缩略图，每一次加载的相册可以从缓存中加载，当有新的图片上传到服务器时，再将缓存更新。

再比如读取一个大文件时，可将文件从磁盘拆分读取输出到网络缓冲区。否则需要读取整个大文件，客户可能不知道要过多久才能下载，并可能中断应用程序的响应。

除了常见的问题和重构策略外，请牢记，我们正在谈论的效率不仅仅是重构了事，也要有优化的兼顾。

10.7.1 网络带宽的效率

1. 从服务器请求资源是否在较低限度

一个 Web 应用程序是一些业务逻辑，例如用 PHP 编写的脚本，输出 HTML、CSS 和 JavaScript 到客户端浏览器。

实际情况上，总有一个用户会为你的页面加载而“等待”，无论他是愿意等待得再久一点，还是发现加载很慢，而关闭窗口跑掉。一般来说，页面在 512KB 的连接速率上，超过 5 秒还没有显示完成，用户将开始感到不耐烦。Amazon 有一个统计数字：如果页面在 3 秒内未加载完成，将留失 57% 的用户，结果是 PV 减少，用户率降低。每延长 1 秒，会损失 16 亿美金。

为了能够快速加载，你需要减少从服务器请求资源的数量。比如分离 CSS 文件，而不是将它们合并在一个文件中。另外 JavaScript 也和 CSS 一样，也需进一步来优化。

我们还可以压缩 CSS 和 JavaScript 文件的尺寸，使用 JavaScript 压缩工具，如 YUI JS 压缩机和 CSSO。对于图片可以使用 CSS sprites 的技术，这种技术可以将一组图片合并在一起成为一个单一的文件，可以有效减少带宽占用。

2. 使用 Ajax

现代化的 Web 应用基于 Ajax 的异步调用功能越来越多，这没有任何问题，只要你的服务器回调时没有太大延迟即可。

但需要注意的是，不要过度使用 Ajax 来执行非常简单的任务，没有必要尽量不去调用服务器，如果真的需要，再去取，而不是持续的实时数据。如果需要像股票类的实时数据，可以考虑使用数据 push 技术，如 Comet 和 WebSocket。

10.7.2 内存效率低

1. 是否使用了递归

递归在开发时确实功能强大，在解决重复问题时省去我们很多麻烦，比如树的遍历和搜索。但在语言层面，特别是 PHP，使用递归是有负面代价的：它会比较占用内存空间。

如果你不知道在做什么，那尽量不用递归解决，换成正常的逻辑算法吧，这种替代递归的代码对内存占用的减少是值得的。

2. 重复太多了吗

使用递归迭代的局限性我们已经了解，另外涉及代码效率，比如嵌套循环，可以使

用其他解决方案来解决。比如我们的页面要显示 20 条新闻文章，可以先显示最新的 5 条，再加一个更多的链接。类似于这种问题，需要多动脑筋，来加速页面加载。

3. 检查数据库查询

数据库的查询延迟是 PHP 应用中很常见的“瓶颈”，因此我们应该总是对查询进行评估计划。比如开发者中有多少人做了这样的偷懒事情：

```
SELECT * FROM users
```

这个查询很简单，但问题是当用户表超过 10 000 个时，查询有很多列，这个 SQL 在一个查询数据库的所有行！

这是一个典型的无计划的查询。如果你没有使用 FROM 子句，你应该花费一些时间来规划需要的 SQL 查询，需要决定什么样条件的数据需要查询。

另外随着记录成倍的增长，我们应该提前考虑到这一点，不要将 * 号看作是理所当然的。要限制特定的字段和行数，如果实在不了解的话，也要使用 LIMIT 子句限制生成的记录集。

10.7.3　程序处理效率低下

1. 不必要的动态语言特性

PHP 有很多的动态特性，像超载的 getter 和 setter 方法，动态的实例化和方法 / 函数调用，还有最危险的 eval() 函数。

有时使用这些语句确实挺有意思，也会感觉是很神奇的元素，以不规范的问题实现一些有创意的解决方案，但一般来说，我们不需要在标准的 Web 应用中使用任何不广泛的代码，不经意也没必要地增加了复杂性和安全问题。

2. 保持良好的编码习惯

这是初学者遇到的常见问题。我们常常对数组进行处理，根据数据的长度做循环处理。有时候常常在每次循环时都做长度处理，而此时只计算一次就可以了。这一规则适合任何函数调用。

我们看如下的代码：

```
$a = array(); //Imagine several keys and pairs.
for($i = 0; $i < count($a); $i++){
    // Do something
}
```

这个循环有什么错误？其实这样的错误相信你可能也遇到过，你可以理解为 1 岁孩子的错误，但有可能你会犯，这是非常低效的循环。每一次循环就会执行一次 count() 函数，并计算 $a 的值。这在开发中是一个很容易落入的圈套，开发人员经常犯的错误是重复计算的信息，在循环中，无论是在循环的定义或是在内部执行的代码循环。

更好地执行函数内执行循环的定义外循环。这减少了循环运行的时间，并且因此减少了函数被调用的次数。这为我们提供了一个瞬时优化。如果我们要重写这段代码可以更好的，我们的代码应该改为如下所示：

```
$a = array();
$count = count($a);
for($i = 0; $i < $count $i++){
    // 代码省略
}
```

我们其实都知道，但往往会在编码时忘记或优化它。总结一句，减少循环中调用函数的数量，就是提升代码性能最简单且有效的方式。

10.8 本章小结

编码规范与代码重构是软件开发的一项重要技术，它可以确保我们的代码质量。

本章前半部分着重讲述“道”的层面——也就是理论层面的深层考虑，后面以代码和实际研发中——“术”的层面讲述规约的执行标准。

应用程序的效率调优是要综合起来做的事，这里建议的准则是编码前优化，以减少系统“瓶颈”为重点，是减少使用低效率分析工具的省力方法。

代码重构是渐进的过程，在小步中逐步推进，不一定在一个时间做到全部。在本章中，我介绍了一套实用的方法可以让代码重构更具价值，比如什么是好代码，如何提高代码可读性、可扩展性和效率。当对代码质量非常的看重时，实际上我们的能力和素质也会随之提高。

曾经有人问我如何学习PHP，感觉自己遇到了瓶颈，相信不少人都有类似的困惑。学习就像在大海中行船，在黑夜时，需要灯塔指引。杜江有十多年的PHP开发经验，有深厚的技术功底。本书面向的对象是有一定PHP基础，但是又想深入学习的PHP开发者。书中不仅讲解了OOP、设计模式，也讲解了如何进行性能优化和调试。甚至，还对于语言之外的代码重构进行了讲解。相信这本书会为你学习PHP带来良好的启发。

——信海龙（苍龙） 阿里巴巴技术专家

杜江与我是同乡、校友，更是心照神交的好兄弟。相识十余载，我对他在互联网技术领域知根究底、传经布道的精神和做法，看在眼中，赞在心中。他是一位从一线磨砺出来的技术高手，在赶集网、今日头条、好乐买、21CTO、正和岛做过技术合伙人或CTO。他对电商、移动互联网技术，有着非同一般的洞见。以他介入互联网技术的广度和深度，我相信他能够在今日取得的成就之上，为行业以及社会做更大的贡献，取得更大的成就。相信PHP开发者读过这本“心法”后，一定会有新的启发，尤其会在技术解决方案和架构模式设计上获益良多。

——贾海冬 车友宝CTO

与杜江相识多年，但一直只闻其美名，却未见真身。他在PHP领域持续专注地投入十余年，服务过不少知名企业，积累了丰富的开发经验。同时，他还为国内PHP技术领域撰写了两本优秀的PHP著作。杜江秉承一贯严谨、负责的作风，创作了这本良心之作，相信能为广大PHP开发者们带去帮助。

——王福强 前挖财首席架构师

上架指导：计算机\程序设计\PHP
ISBN 978-7-111-54796-9
9 787111 547969
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259
华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn