Notes Collection of PHP

PHP

THEQIONG.COM

穷屌丝联盟

PHP Notes

穷屌丝联盟

2017 年 4 月 25 日

目录

插图

表格

Part I

Introduction

Chapter 1

Overview

PHP[1]是一种开源的通用计算机脚本语言，尤其适用于网络开发并可嵌入 HTML 中使用。

PHP 针对大多数 HTTP 服务器都提供了相应的模块，而且 PHP 支持 CGI 标准，因此 PHP 可以作为 CGI 处理器来使用。

PHP 开始注重程序运行性能和自身代码的模块性之后，PHP 无疑成为了另一种高效率的选择。例如，PHP 内置的标准库（SPL）提供了一组类和接口来支持常用的数据结构类（例如堆栈、队列和堆等）以及遍历这些数据结构的迭代器，或者用户可以自己实现 SPL 接口。

PHP 现在由 PHP Group 和开放源代码社区维护 PHP 标准，并以 PHP License 作为许可协议，限制了 PHP 名称的使用，和 GPL（开放源代码许可协议）不兼容。

“.php” 是 PHP 的默认扩展名，所有以.php 结尾的文件都将由 PHP 来处理，只需要把.php 文件部署到支持 PHP 的服务器上的 Web 目录中，服务器将自动解析这些文件，无需编译也不用安装任何其它的工具，仅仅只需把这些使用了 PHP 的文件想象成简单的 HTML 文件，其中只不过多了一种新的标识符，在这里可以做各种各样的事情。

PHP 的语法借鉴了 C、Java 和 Perl 等语言，主要目标是允许网络开发人员快速编写动态页面。

PHP 针对不同层次的用户提供了相关的特性（包括高级特性），因此尽管 PHP 的开发是以服务端脚本为主，也可以被用于其他很多领域。

PHP 的跨平台特性使其可以在多数的服务器和操作系统上运行，而且的到了大多数 Web 服务器的支持，现在的云计算平台（Google App Engine、Amazon EC2 和 Microsoft Azure 等）都提供了对 PHP 的支持。

[1]PHP：Hypertext Preprocessor（超文本预处理器）就是一种递归命名。

另外，PHP 的扩展性使其除了可以向最终用户提供数据库、协议和 API 的基础结构之外，还允许开发者自己加入并提交新的模块。

PHP 的应用范围相当广泛，尤其是在网页程序的开发上。

1. 服务器端程序
    - PHP 解析器（CGI 或服务器模块）
    - Web 服务器（Nginx 和 Apache 等）
    - Web 浏览器
2. 命令行脚本
    - PHP 命令行模式
    - 定时任务（Cron 或 Tast Scheduler）

PHP 丰富的功能并不局限于输出 HTML，还能被用来动态输出图像、PDF 文件、Flash 动画（使用 libswf 和 Ming），用户一般使用 PHP 来收集表单数据、生成动态网页或者发送/接收 Cookies 等。

PHP 在处理 HTML 表单时，表单的任何元素都在 PHP 脚本中自动生效。

```
<form action="action.php" method="post">
 <p>姓名: <input type="text" name="name" /></p>
 <p>年龄: <input type="text" name="age" /></p>
 <p><input type="submit" /></p>
</form>
```

当用户填写了该表单并点击了提交按钮后，页面 action.php 将被调用。

```
你好, <?php echo htmlspecialchars($_POST['name']); ?>。
你 <?php echo (int)$_POST['age']; ?> 岁了。
```

htmlspecialchars() 使得 HTML 之中的特殊字符被正确的编码，从而不会被使用者在页面注入 HTML 标签或者 Javascript 代码（例如 age 字段），也可以使用 PHP 的 filter 扩展来自动完成该工作，PHP 会自动设置 `$_POST['name']` 和 `$_POST['age']` 变量。

用户也可以在 PHP 中处理 XForms 的输入，尽管用户可能更喜欢使用长久以来支持良好的 HTML 表单。

PHP 可以非常简便的输出文本（例如 XHTML 以及任何其它形式的 XML 文件），PHP 还可以通过在服务端开辟出一块动态内容的缓存从而直接把它们打印出来，或者将它们存储到文件系统中。

PHP 支持使用 Perl 兼容正则表达式（PCRE）以及其他扩展和工具来解析和访问 XML 文档，同时 PHP 还将所有的 XML 功能标准化在 libxml2 库中，并且还增加了 SimpleXML，XMLReader 以及 XMLWriter 来支持并扩充功能。

PHP 可以在开发时选择使用过程和面向对象，或者两者混合的方式来开发。例如，从 PHP 5 开始引入了完全的对象模型，弥补了 PHP 4 对 OOP 支持的不足。

PHP 对数据库的支持很全面，用户可以使用任何针对某数据库的扩展（例如 mysql）编写数据库支持的网页，也可以使用抽象层（例如 PDO）或者通过 ODBC 扩展连接到任何支持 ODBC 标准的数据库。其它一些数据库也可能会用 cURL 或者 sockets（例如 CouchDB）。

PHP 还支持利用 LDAP、IMAP、SNMP、NNTP、POP3、HTTP、COM（Windows 环境）等协议的服务，还可以通过开放原始网络端口来使得任何其它的协议能够协同工作。

PHP 支持和所有 Web 开发语言之间的 WDDX 复杂数据交换，而且 PHP 支持 Java 对象的即时连接，可以透明地将其用作 PHP 对象。

PHP 框架可以使项目得到更快更简单的部署和更加敏捷的开发效率，因此在实际生产中框架的使用已非常普遍，例如 PHP 官方的框架为 Zend framework。

在使用框架进行开发时，绝大部分的上层代码以及项目结构都会基于所使用的框架，因此很多关于设计模式的决定已经由框架提前实现。

1.1 History

PHP 最初是 Rasmus Lerdorf 为了要维护个人网页，而用 C 语言开发的一些用以取代原先使用 Perl 封装的 CGI 工具程序集。这些工具程序用来显示 Rasmus Lerdorf 的个人履历以及统计网页流量。他将这些程序和一些窗体解释器集成起来，称为 PHP/FI，其中 FI 可以和数据库连接，从而让 PHP 可以产生简单的动态网页程序。

Rasmus Lerdorf 在 1995 年 6 月 8 日将 PHP/FI 公开发布，希望可以通过社区来加速程序开发与查找错误。这个发布的版本命名为 PHP 2，已经有今日 PHP 的一些雏型，像是类似 Perl 的变量命名方式、窗体处理功能、以及嵌入到 HTML 中运行的能力。程序语法上也类似 Perl，有较多的限制，不过更简单、更有弹性。

1997 年，Zeev Suraski 和 Andi Gutmans 不满足于 PHP 在大型项目中的表现，从而重写了 PHP 的语法分析器，这成为了 PHP 3 的基础，而 PHP 也在这个时候改称为 PHP: Hypertext Preprocessor。

1997 年 11 月，开发团队发布了 PHP/FI 2，随后就开始 PHP 3 的开放测试，并于 1998 年 6 月正式发布 PHP 3。

Zeev Suraski 和 Andi Gutmans 在 PHP 3 发布后开始改写 PHP 的核心，这个在 1999 年发布的语法分析器称为 Zend Engine，他们也在以色列的 Ramat Gan 成立了 Zend Technologies 来管理 PHP 的开发。

2000 年 5 月 22 日，以 Zend Engine 1.0 为基础的 PHP 4 正式发布，增加了数组操作函数、完整的会话机制、对输出缓存的支持等。

2004 年 7 月 13 日发布的 PHP 5 使用了第二代的 Zend Engine，而且 PHP5 包含了许多新特色，比如强化的面向对象功能、引入 PDO 以及许多性能上的增强。

从 2008 年开始，PHP 5 成为了 PHP 唯一维护中的稳定版本，PHP 4 已经不会继续更新，以鼓励用户迁移到 PHP 5。

PHP 7 的开发正在进行中，主要的改进有 PHPNG、JIT 引擎、抽象语法树编译、异步编程。

要查看当前 PHP 信息，可以通过网页或者命令行，其中通过网页可以直观的显示。

```
<?php
  phpinfo();
?>
```

或者在命令行中输入如下的命令：

```
$ php -i
//or
$ php -r 'phpinfo();'
```

`phpinfo()` 函数可以得到很多有关当前系统的有用信息，例如预定义变量、已经加载的 PHP 模块和配置信息。

为了移除敏感信息，比如 AUTH_USER 和 AUTH_PASSWORD，可以修改代码如下：

```
<?php
  // start output buffering
  ob_start();

  // send phpinfo content
  phpinfo();

  // get phpinfo content
  $html = ob_get_contents();

  // flush the output buffer
  ob_end_clean();

  // remove auth data
  if(isset($_SERVER['AUTH_USER']))
    $html = str_replace($_SERVER['AUTH_USER'], '<i>no value</i>',$html);
  if(isset($_SERVER['AUTH_PASSWORD']))
    $html = str_replace($_SERVER['AUTH_PASSWORD'], '<i>no value</i>', $html);
```

```
  echo $html;
?>
```

为了比较上述两种方法的异同，可以在一个页面中同时运行这两个测试页面：

```
<html>
  <frameset cols="50%,50%">
    <frame src="phptest.php">
    <frame src="phpinfo.php">
  </frameset>
</html>
```

在兼容性方面，PHP 有可能影响到老版本的代码的最重要的两点改动分别是：

- 取消了旧的 $HTTP_*_VARS数组（在函数或者方法中原本是全局变量）。

PHP 4.1.0 版本引入了如下超全局数组变量：

- $_GET

  PHP 保留的超全局变量变量 $_GET 包含所有的 GET 数据。
- $_POST

  PHP 保留的超全局变量 $_POST 包含所有的 POST 数据。
- $_COOKIE

  PHP 保留的超全局变量 $_COOKIE 包含所有的 COOKIE 数据。
- $_SERVER

  PHP 保留的超全局变量 $_SERVER 包含 Web 服务器提供的所有信息。
- $_FILES

  PHP 保留的超全局变量 $_FILES 包含所有的 FILES 数据。
- $_ENV

  PHP 保留的超全局变量 $_ENV 包含所有的 ENV 数据。
- $_REQUEST

  PHP 保留的超全局变量 $_REQUEST 包含所有的 GET、POST 和 COOKIE 的数据。
- $_SESSION

  PHP 保留的超全局变量 $_SESSION 包含所有的 SESSION 数据。

有些 $HTTP_*_VARS数组（诸如 $HTTP_POST_VARS 等）从 PHP 3 就已经开始使用，从 PHP 5.4.0 开始将不再有效。

自 PHP 5.0.0 起, 用 register_long_arrays 设置选项可禁用长类型的 PHP 预定义变量数组。

- 外部变量不再被默认注册为全局变量。也就是说，从 PHP 4.2.0 版开始，php.ini 中的设置选项 register_globals 默认值变成了 off。

建议用以上提到的超全局数组变量来访问这些值。

如果该选项被设置为 on，则可以在http://www.example.com/foo.php?id=42中直接使用变量 $id。不过，不管被设置为 on 还是 off，$_GET['id'] 一直有效。

1.2 Dynamic Parser

无论以模块还是 CGI 的形式安装 PHP，PHP 的解释器都可以在服务器上执行访问文件、运行命令以及创建网络连接等操作，而且这些操作可能产生不安全隐患，因此需要正确地安装和配置 PHP 以及编写安全的代码。

PHP 提供的 CLI SAPI 不会将当前目录改为已运行的脚本所在的目录。例如，下面示例可以反映 CLI SAPI 模块与 CGI SAPI 模块之间的不同：

```
<?php
// 名为 test.php 的简单测试程序
echo getcwd(), "\n";
```

CGI SAPI 将当前目录修改为运行 PHP 脚本所在的目录，因此输出结果如下：

```
$ pwd
/tmp
$ php-cgi -f another_directory/test.php
/tmp/another_directory
```

CLI SAPI 将使用当前的目录，因此输出结果如下：

```
$ pwd
/tmp
$ php -q another_directory/test.php
/tmp
```

用户在命令行运行 PHP CGI 应用程序时可以给 CGI SAPI 加上 -C 参数来使其支持 CLI SAPI 的功能。

CLI SAPI 模块有以下三种不同的方法来获取要运行的 PHP 代码：

1. 使用 PHP 运行指定文件

   ```
   $ php my_script.php
   $ php -f my_script.php
   ```

   CLI SAPI 指定的 PHP 脚本并非必须要以.php 为扩展名，可以有任意的文件名和扩展名来运行任何文件。

2. 使用 PHP 在命令行直接运行 PHP 代码

   ```
   $ php -r 'print_r(get_defined_constants());'
   ```

3. 使用 PHP 运行通过标准输入（stdin）提供的 PHP 代码

```
$ some_application | some_filter | php | sort -u >final_output.txt
```

CLI SAPI 可以动态地生成 PHP 代码并通过命令行运行这些代码。

以上三种运行代码的方法不能同时使用。

1.3 Template Engine

原生 PHP 模板就是指直接用 PHP 来写模板，这是很自然的选择，因为 PHP 本身其实是个模板语言。这代表你可以在其他的语言（比如 HTML）中结合 PHP 使用。这对 PHP 开发者相当有利，因为不需要额外学习新的语法，他们熟知可以使用的函数，并且使用的编辑器也已经内置了语法高亮和自动补全。

原生的 PHP 模板没有了 Blade 等模板引擎的编译阶段，速度会更快。

现今的 PHP 框架都会使用一些模板系统，其中大多数是使用原生的 PHP 语法。在框架之外，一些类库（比如 Plates 或 Aura.View）都提供了现代化模板的常见功能，比如继承、布局、扩展，让原生的 PHP 模板更容易使用。

- Aura.View(https://github.com/auraphp/Aura.View)
- Blade(https://github.com/laravel/laravel)
- Brainy(https://github.com/box/brainy)
- Dwoo(https://github.com/dwoo-project/dwoo)
- Latte(https://github.com/nette/latte)
- Mustache(https://github.com/bobthecow/mustache.php)
- PHPTAL(https://github.com/phptal/PHPTAL)
- Plates(https://github.com/thephpleague/plates)
- Smarty(https://github.com/smarty-php/smarty/)
- Twig(https://github.com/twigphp/Twig)
- Zend\View(https://github.com/zendframework/zend-view)

1.3.1 Plates

```
<?php // user_profile.php ?>

<?php $this->insert('header', ['title' => 'User Profile']) ?>

<h1>User Profile</h1>
<p>Hello, <?=$this->escape($name)?></p>
```

```
<?php $this->insert('footer') ?>
```

Example 1 原生 PHP 模板使用 Plates 类库实现继承

```
<?php // template.php ?>

<html>
<head>
   <title><?=$title?></title>
</head>
<body>

<main>
   <?=$this->section('content')?>
</main>

</body>
</html>
```

```
<?php // user_profile.php ?>

<?php $this->layout('template', ['title' => 'User Profile']) ?>

<h1>User Profile</h1>
<p>Hello, <?=$this->escape($name)?></p>
```

1.3.2 Smarty

尽管 PHP 不断升级为成熟的、面向对象的语言，但是作为模板语言的特性并没有改善多少。编译模板（比如 Twig 或 Smarty）提供了模板专用的新语法，填补了这片空白。

从自动转义[2]到继承以及简化控制结构，编译模板被设计地更容易编写，可读性更高，同时使用上也更加的安全。

编译模板也可以在不同的语言中使用（例如 Mustache），虽然这些编译模板时需要在性能上产生一些轻微的影响，不过如果适当的使用缓存，影响就变得非常小了。

Example 2 使用 Twig 类库编译模板

[2] 虽然 Smarty 提供了自动转义的功能，不过这个功能默认是关闭的。

```
{% include 'header.html' with {'title': 'User Profile'} %}

<h1>User Profile</h1>
<p>Hello, {{ name }}</p>

{% include 'footer.html' %}
```

Example 3　使用 Twig 类库实现继承

```
// template.html

<html>
<head>
    <title>{% block title %}{% endblock %}</title>
</head>
<body>

<main>
    {% block content %}{% endblock %}
</main>

</body>
</html>
```

```
// user_profile.html

{% extends "template.html" %}

{% block title %}User Profile{% endblock %}
{% block content %}
    <h1>User Profile</h1>
    <p>Hello, {{ name }}</p>
{% endblock %}
```

1.4　Dynamic Tracing

DTrace 跟踪调试框架永远可用，并且额外消耗极低。

DTrace 可以跟踪操作系统行为和用户程序的执行情况，可以显示参数值，也可以用来分析性能问题。例如，用户可以使用 DTrace D 脚本语言创建脚本文件来监控探针来高效地分析数据指针。

除非用户使用 DTrace D 脚本激活并监控 PHP 探针，否则它并不会运行，所以不会给正常的应用执行带来任何性能损耗。即使 PHP 探针被激活，它的消耗也是非常低的，甚至可以直接在生产系统中使用。

在运行时可以激活 PHP “用户级静态定义跟踪”（USDT）探针。举例来说，如果 D 脚本正在监控 PHP function-entry 探针，那么，每当 PHP 脚本函数被调用的时候，这个探针将被触发，同时 D 脚本中所关联的动作代码将被执行。在 D 脚本的动作代码中，可以打印出诸如函数所在源文件等信息的探针参数，也可以记录诸如函数执行次数这样的聚合数据。

虽然不同平台提供的 DTrace 功能并不完全相同，不过 DTrace 一般都可以跟踪函数调用以及操作系统行为等。

`--enable-dtrace` 配置参数可以来开启 PHP 核心的静态探针，PHP 扩展可以拥有额外的静态探针，不过提供了自有探针的 PHP 扩展需要分别构建。

```
$ ./configure --enable-dtrace
$ make
$ make install
```

表 1.1: PHP DTrace 静态探针

探针名称	探针描述	探针参数
request-startup	请求开始时触发。	`char *file, char *request_uri, char *request_method`
request-shutdown	请求关闭时触发。	`char *file, char *request_uri, char *request_method`
compile-file-entry	脚本开始编译时触发。	`char *compile_file, char *compile_file_translated`
compile-file-return	脚本完成编译时触发。	`char *compile_file, char *compile_file_translated`

探针名称	探针描述	探针参数
execute-entry	操作数数组开始执行时触发，例如函数调用、文件包含以及生成器恢复时会被触发。	`char *request_file, int lineno`
execute-return	操作数数组执行完毕之后触发。	`char *request_file, int lineno`
function-entry PHP	引擎进入PHP函数或者方法调用时触发。	`char *function_name, char *request_file, int lineno, char *classname, char *scope`
function-return	PHP引擎从PHP函数或者方法调用返回后触发。	`char *function_name, char *request_file, int lineno, char *classname, char *scope`
exception-thrown	有异常抛出时触发。	`char *classname`
exception-caught	有异常被捕获时触发。	`char *classname`
error	无论error_reporting的设定如何，在发生错误时都会触发。	`char *errormsg, char *request_file, int lineno`

要列出PHP中可用的静态探针，开启一个PHP进程，然后执行 `dtrace -l`，输出结果如下：

```
$ dtrace -l
ID   PROVIDER         MODULE                          FUNCTION NAME
[ . . . ]
4    php15271           php             dtrace_compile_file compile-file-entry
5    php15271           php             dtrace_compile_file compile-file-return
6    php15271           php                       zend_error error
7    php15271           php  ZEND_CATCH_SPEC_CONST_CV_HANDLER exception-caught
8    php15271           php    zend_throw_exception_internal exception-thrown
9    php15271           php               dtrace_execute_ex execute-entry
10   php15271           php          dtrace_execute_internal execute-entry
11   php15271           php               dtrace_execute_ex execute-return
12   php15271           php          dtrace_execute_internal execute-return
13   php15271           php               dtrace_execute_ex function-entry
14   php15271           php               dtrace_execute_ex function-return
15   php15271           php             php_request_shutdown request-shutdown
16   php15271           php              php_request_startup request-startup
```

如果运行的是 Apache web 服务器，那么模块名称可能是 libphp5.so，并且可能会出现多块信息，每个运行中的 Apache 进程对应一个输出块。

- Provider 列由 PHP 和当前进程 id 的组成。
- Function Name 列表示 Provider 对应的 PHP 内部 C 实现函数名称。

如果没有运行任何 PHP 进程，那么就不会显示任何 PHP 探针。

下面的示例展示了基本的 DTrace D 脚本。

Example 4 PHP 静态探针

```
#!/usr/sbin/dtrace -Zs

#pragma D option quiet

php*:::compile-file-entry
{
    printf("PHP compile-file-entry\n");
    printf(" compile_file          %s\n", copyinstr(arg0));
    printf(" compile_file_translated %s\n", copyinstr(arg1));
}

php*:::compile-file-return
{
    printf("PHP compile-file-return\n");
    printf(" compile_file          %s\n", copyinstr(arg0));
```

```
    printf(" compile_file_translated %s\n", copyinstr(arg1));
}

php*:::error
{
    printf("PHP error\n");
    printf(" errormsg              %s\n", copyinstr(arg0));
    printf(" request_file          %s\n", copyinstr(arg1));
    printf(" lineno                %d\n", (int)arg2);
}

php*:::exception-caught
{
    printf("PHP exception-caught\n");
    printf(" classname             %s\n", copyinstr(arg0));
}

php*:::exception-thrown
{
    printf("PHP exception-thrown\n");
    printf(" classname             %s\n", copyinstr(arg0));
}

php*:::execute-entry
{
    printf("PHP execute-entry\n");
    printf(" request_file          %s\n", copyinstr(arg0));
    printf(" lineno                %d\n", (int)arg1);
}

php*:::execute-return
{
    printf("PHP execute-return\n");
    printf(" request_file          %s\n", copyinstr(arg0));
    printf(" lineno                %d\n", (int)arg1);
}

php*:::function-entry
{
    printf("PHP function-entry\n");
    printf(" function_name         %s\n", copyinstr(arg0));
```

```
    printf(" request_file          %s\n", copyinstr(arg1));
    printf(" lineno                %d\n", (int)arg2);
    printf(" classname             %s\n", copyinstr(arg3));
    printf(" scope                 %s\n", copyinstr(arg4));
}

php*:::function-return
{
    printf("PHP function-return\n");
    printf(" function_name         %s\n", copyinstr(arg0));
    printf(" request_file          %s\n", copyinstr(arg1));
    printf(" lineno                %d\n", (int)arg2);
    printf(" classname             %s\n", copyinstr(arg3));
    printf(" scope                 %s\n", copyinstr(arg4));
}

php*:::request-shutdown
{
    printf("PHP request-shutdown\n");
    printf(" file                  %s\n", copyinstr(arg0));
    printf(" request_uri           %s\n", copyinstr(arg1));
    printf(" request_method        %s\n", copyinstr(arg2));
}

php*:::request-startup
{
    printf("PHP request-startup\n");
    printf(" file                  %s\n", copyinstr(arg0));
    printf(" request_uri           %s\n", copyinstr(arg1));
    printf(" request_method        %s\n", copyinstr(arg2));
}
```

此脚本在 dtrace 命令中使用了 -Z 选项，此选项保证即使在没有任何 PHP 进程运行的时候脚本也能够正确执行。如果省略了此选项，当没有任何探针可监控的时候，脚本会立即终止执行。

在运行此脚本的过程中，它将监控全部 PHP 核心静态探针。

在下面的示例，运行 D 脚本后再运行一个 PHP 脚本或者 PHP 应用，用来进行监控的 D 脚本就会输出每个探针被触发时所携带的参数。

```
# ./all_probes.d
```

在多 CPU 的主机上，探针的显示顺序可能不是连续的，主要取决于哪颗 CPU 执行探针以及多个 CPU 之间的线程迁移情况，可以通过显示探针时间戳来减少混淆。

```
php*:::function-entry
{
     printf("%lld: PHP function-entry ", walltimestamp);
     [ . . .]
}
```

在某些 Linux 发行版中，可以使用 SystemTap 工具来监控 PHP 内置的 DTrace 静态探针，或者可以配置 PHP 打开 DTrace 静态探针。例如，在 Oracle Linux 系统上启动 UEK3 内核，并进行如下操作可以打开 DTrace 静态探针。

```
# yum install systemtap-sdt-devel
# ./configure --enable-dtrace ...
# make
# make install
# modprobe fasttrap
# chmod 666 /dev/dtrace/helper
```

除了 chmod 命令，也可以使用 ACL 包规则来限制特定用户对于设备的访问权限。

stap 命令可以列出 PHP 静态探针，例如：

```
# stap -l 'process.provider("php").mark("*")' -c 'sapi/cli/php -i'
```

输出如下：

```
process("sapi/cli/php").provider("php").mark("compile__file__entry")
process("sapi/cli/php").provider("php").mark("compile__file__return")
process("sapi/cli/php").provider("php").mark("error")
process("sapi/cli/php").provider("php").mark("exception__caught")
process("sapi/cli/php").provider("php").mark("exception__thrown")
process("sapi/cli/php").provider("php").mark("execute__entry")
process("sapi/cli/php").provider("php").mark("execute__return")
process("sapi/cli/php").provider("php").mark("function__entry")
process("sapi/cli/php").provider("php").mark("function__return")
process("sapi/cli/php").provider("php").mark("request__shutdown")
process("sapi/cli/php").provider("php").mark("request__startup")
```

Example 5　使用 SystemTap 追踪 PHP 静态探针

```
probe process("sapi/cli/php").provider("php").mark("compile__file__entry") {
    printf("Probe compile__file__entry\n");
    printf(" compile_file %s\n", user_string($arg1));
```

```
    printf(" compile_file_translated %s\n", user_string($arg2));
}
probe process("sapi/cli/php").provider("php").mark("compile__file__return")
    {
    printf("Probe compile__file__return\n");
    printf(" compile_file %s\n", user_string($arg1));
    printf(" compile_file_translated %s\n", user_string($arg2));
}
probe process("sapi/cli/php").provider("php").mark("error") {
    printf("Probe error\n");
    printf(" errormsg %s\n", user_string($arg1));
    printf(" request_file %s\n", user_string($arg2));
    printf(" lineno %d\n", $arg3);
}
probe process("sapi/cli/php").provider("php").mark("exception__caught") {
    printf("Probe exception__caught\n");
    printf(" classname %s\n", user_string($arg1));
}
probe process("sapi/cli/php").provider("php").mark("exception__thrown") {
    printf("Probe exception__thrown\n");
    printf(" classname %s\n", user_string($arg1));
}
probe process("sapi/cli/php").provider("php").mark("execute__entry") {
    printf("Probe execute__entry\n");
    printf(" request_file %s\n", user_string($arg1));
    printf(" lineno %d\n", $arg2);
}
probe process("sapi/cli/php").provider("php").mark("execute__return") {
    printf("Probe execute__return\n");
    printf(" request_file %s\n", user_string($arg1));
    printf(" lineno %d\n", $arg2);
}
probe process("sapi/cli/php").provider("php").mark("function__entry") {
    printf("Probe function__entry\n");
    printf(" function_name %s\n", user_string($arg1));
    printf(" request_file %s\n", user_string($arg2));
    printf(" lineno %d\n", $arg3);
    printf(" classname %s\n", user_string($arg4));
    printf(" scope %s\n", user_string($arg5));
}
probe process("sapi/cli/php").provider("php").mark("function__return") {
```

```
    printf("Probe function__return: %s\n", user_string($arg1));
    printf(" function_name %s\n", user_string($arg1));
    printf(" request_file %s\n", user_string($arg2));
    printf(" lineno %d\n", $arg3);
    printf(" classname %s\n", user_string($arg4));
    printf(" scope %s\n", user_string($arg5));
}
probe process("sapi/cli/php").provider("php").mark("request__shutdown") {
    printf("Probe request__shutdown\n");
    printf(" file %s\n", user_string($arg1));
    printf(" request_uri %s\n", user_string($arg2));
    printf(" request_method %s\n", user_string($arg3));
}
probe process("sapi/cli/php").provider("php").mark("request__startup") {
    printf("Probe request__startup\n");
    printf(" file %s\n", user_string($arg1));
    printf(" request_uri %s\n", user_string($arg2));
    printf(" request_method %s\n", user_string($arg3));
}
```

在 PHP 脚本的执行过程中，上述脚本会跟踪所有的 PHP 核心静态探针：

```
# stap -c 'sapi/cli/php test.php' all_probes.stp
```

1.5 Garbage Collection

每个 php 变量存在一个叫"zval" 的变量容器中。

一个 zval 变量容器，除了包含变量的类型和值，还包括两个字节的额外信息。

第一个是"is_ref"，本身是一个用来标识这个变量是否是属于引用集合 (reference set) 的 bool 值。通过这个字节，php 引擎才能把普通变量和引用变量区分开来。

PHP 允许用户通过使用 & 来使用自定义引用，因此 zval 变量容器中还有一个内部引用计数机制来优化内存使用。

第二个额外字节是"refcount"，用以表示指向这个 zval 变量容器的变量 (也称符号即 symbol) 个数，所有的符号存在一个符号表中，其中每个符号都有作用域 (scope)，那些主脚本（例如通过浏览器请求的的 PHP 脚本）和每个函数或者方法也都有作用域。

当一个变量被赋常量值时，就会生成一个 zval 变量容器。

Example 6　生成一个新的 zval 容器

```
<?php
$a = "new string";
```

上述 PHP 脚本中定义了新的变量 a，是在当前作用域中生成的，同时还生成了类型为 string 和值为 new string 的变量容器，其额外的两个字节的信息如下：

- `"is_ref"` 被默认设置为 FALSE，因为没有任何自定义的引用生成。
- `"refcount"` 被设定为 1，因为这里只有一个变量使用这个变量容器。

当`"refcount"` 的值是 1 时，`"is_ref"` 的值总是 FALSE，使用 Xdebug 提供的函数 xdebug_debug_zval() 可以显示`"refcount"` 和`"is_ref"` 的值。

Example 7 显示 zval 信息

```
<?php
xdebug_debug_zval('a');
```

上述示例的结果如下：

```
a: (refcount=1, is_ref=0)='new string'
```

把一个变量赋值给另一变量将增加引用次数 (refcount)，变量容器在”refcount“变成 0 时就被销毁，而且当任何关联到某个变量容器的变量离开它的作用域（例如函数执行结束）或者对变量调用了函数 unset() 时，”refcount“就会减 1。

Example 8 增加一个 zval 的引用计数

```
<?php
$a = "new string";
$b = $a;
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=2, is_ref=0)='new string'
```

同一个变量容器被变量 a 和变量 b 关联时，如果没有必要，PHP 就不会去复制已生成的变量容器。

Example 9 减少一个 zval 的引用计数

```
<?php
$a = "new string";
$c = $b = $a;
```

```
xdebug_debug_zval( 'a' );
unset( $b, $c );
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=3, is_ref=0)='new string'
a: (refcount=1, is_ref=0)='new string'
```

如果现在执行 `unset($a);`，那么包含类型和值的这个变量容器就会从内存中删除。

PHP 的复合数据类型（例如 array 和 object）的值与标量 (scalar) 类型的值不同，array 和 object 类型的变量把它们的成员或属性存在自己的符号表中。例如，下面的例子将生成三个 zval 变量容器：

Example 10　生成一个 array zval

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=1, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42
)
```

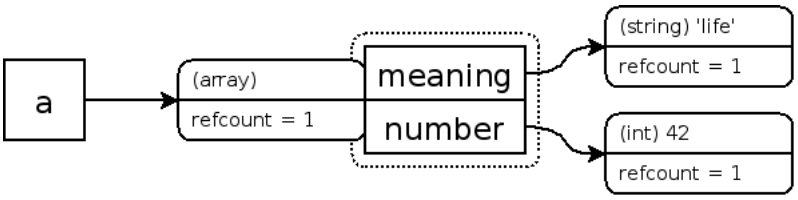

图 1.1: array zval

这里的 array 生成的三个 zval 变量容器分别是 a、meaning 和 number，增加和减少”refcount”的规则和上面提到的相同。

下面的示例在数组中再添加一个元素，并且把它的值设为数组中已存在元素的值：

Example 11　添加一个已经存在的元素到数组中

```php
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=2, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42,
   'life' => (refcount=2, is_ref=0)='life'
)
```

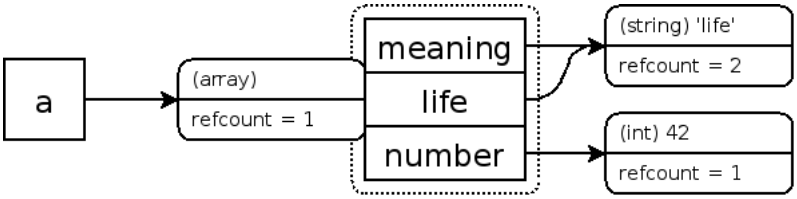

图 1.2: 修改 array zval

从上面的 xdebug 输出信息可以看到原有的数组元素和新添加的数组元素关联到同一个”refcount=2” 的 zval 变量容器.

尽管 Xdebug 的输出显示两个值为’life’ 的 zval 变量容器，其实是同一个，xdebug_debug_zval() 不显示这个信息，不过还是能通过显示内存指针信息来看到。

删除数组中的一个元素类似于从作用域中删除一个变量，删除后的数组中的这个元素所在的容器的“refcount”值减少。同样地，当“refcount”为 0 时，这个变量容器就从内存中被删除。

Example 12 从数组中删除一个元素

```php
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
unset( $a['meaning'], $a['number'] );
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=1, is_ref=0)=array (
   'life' => (refcount=1, is_ref=0)='life'
)
```

在添加一个数组本身作为这个数组的元素时，需要加入引用操作符，否则 PHP 将生成一个复制。

Example 13　把数组作为一个元素添加到自己

```
<?php
$a = array( 'one' );
$a[] =& $a;
xdebug_debug_zval( 'a' );
```

上述示例的结果如下：

```
a: (refcount=2, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=2, is_ref=1)=...
)
```

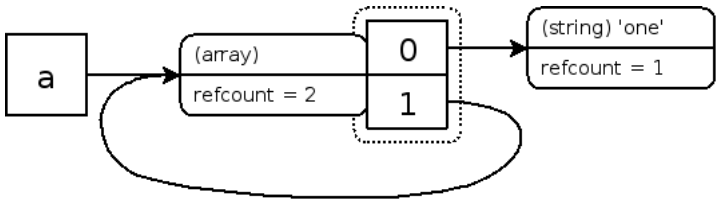

图 1.3: 把数组作为一个元素添加到自己

数组变量 (a) 同时也是这个数组的第二个元素 (1) 指向的变量容器中“refcount”为 2，上面的输出结果中的"..." 说明发生了递归操作，显然在这种情况下意味着"..." 指向原始数组。

对一个变量调用 unset 将删除这个符号，而且它指向的变量容器中的引用次数也减 1，所以如果在执行完上面的代码后对变量 $a 调用 unset，那么变量 $a 和数组元素"1" 所指向的变量容器的引用次数减 1，即从"2" 变成"1"。

Example 14　删除数组元素

```
(refcount=1, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=1, is_ref=1)=...
)
```

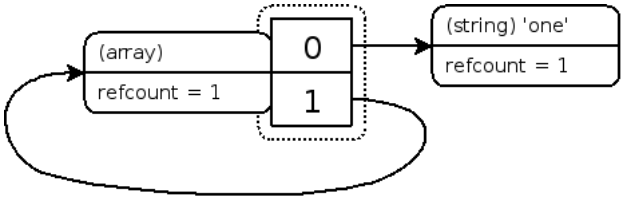

图 1.4: 删除数组元素

另外，尽管不再有某个作用域中的任何符号指向这个结构 (就是变量容器)，由于数组元素“1”仍然指向数组本身，所以这个容器不能被清除。事实上，如果没有其他的符号指向它，用户没有办法清除这个结构，结果就会导致内存泄漏。

PHP 通过在脚本执行结束时清除这个数据结构来避免内存泄露，但是在 PHP 清除之前需要耗费不少内存。如果要实现分析算法，或者要做其他像一个子元素指向它的父元素这样的操作，这种情况就会经常发生。

上述同样的情况也会发生在对象上，实际上对象更有可能出现这种情况，因为对象总是隐式的被引用，结果就是上面这样的情况发生仅仅一两次可能没什么问题，但是如果出现多次的内存泄漏，结果就显然是个大问题。

这样的问题往往发生在长时间运行的脚本中，比如请求基本上不会结束的守护进程 (deamons) 或者单元测试中的大的套件 (sets) 中。其中，后者的例子就是在给巨大的 eZ 组件库的模板组件执行单元测试就可能会出现问题，有时测试可能需要耗用 2GB 的内存，而测试服务器很可能没有这么大的内存。

总而言之，引用计数内存机制无法处理循环的引用内存泄漏，需要考虑使用引用计数系统中的同步周期回收中的同步算法来处理这个内存泄漏问题。

1.5.1 Reference Counting

首先，我们先要建立一些基本规则，如果一个引用计数增加，它将继续被使用，当然就不再在垃圾中。如果引用计数减少到零，所在变量容器将被清除 (free)。就是说，仅仅在引用计数减少到非零值时，才会产生垃圾周期 (garbage cycle)。

其次，在一个垃圾周期中，通过检查引用计数是否减 1，并且检查哪些变量容器的引用次数是零，来发现哪部分是垃圾。

在步骤 A 中，为避免不得不检查所有引用计数可能减少的垃圾周期，这个算法把所有可能根 (possible roots 都是 zval 变量容器) 都放在根缓冲区 (root buffer) 中 (用紫色来标记，称为疑似垃圾)，这样可以同时确保每个可能的垃圾根 (possible garbage root) 在缓冲区中只出现一次。

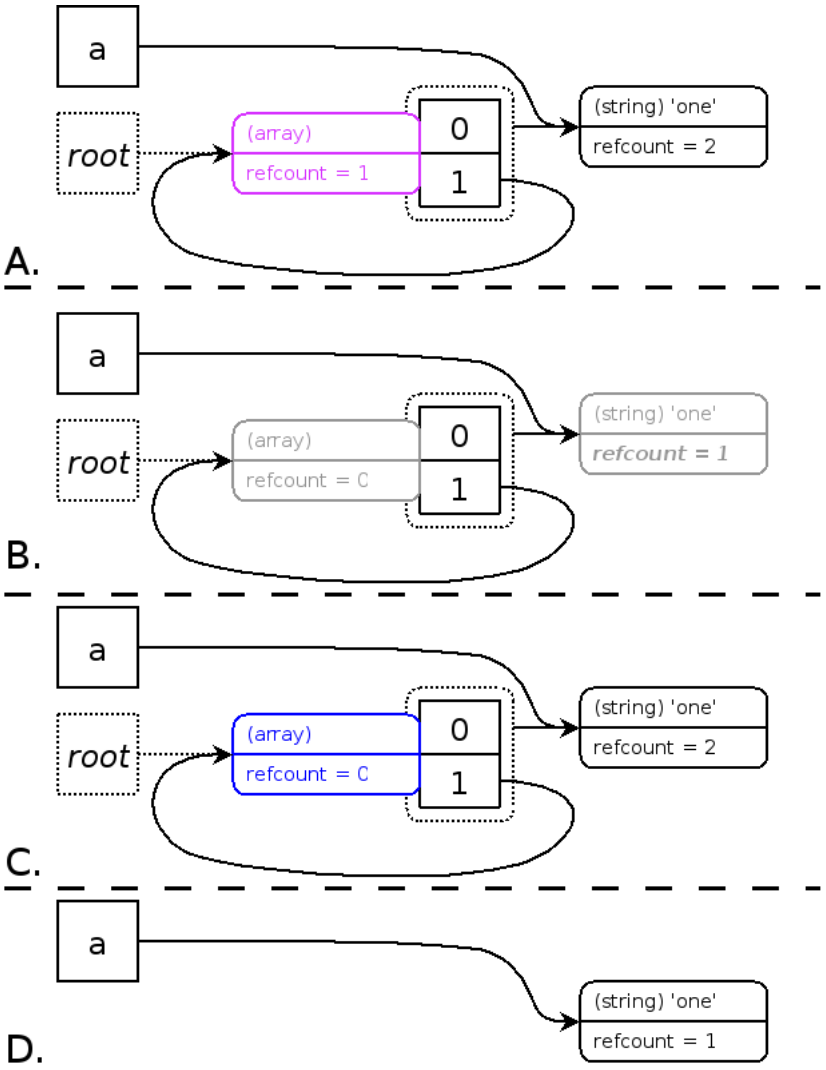
a
root
(array)
refcount = 1
0
1
(string) 'one'
refcount = 2
A.
a
root
(array)
refcount = 0
0
1
(string) 'one'
refcount = 1
B.
a
root
(array)
refcount = 0
0
1
(string) 'one'
refcount = 2
C.
a
(string) 'one'
refcount = 1
D.

图 1.5: 内存回收算法

仅仅在根缓冲区满了时，才对缓冲区内部所有不同的变量容器执行垃圾回收操作。

在步骤 B 中，模拟删除每个紫色变量。模拟删除时可能将不是紫色的普通变量引用数减 1，如果某个普通变量引用计数变成 0 了，就对这个普通变量再做一次模拟删除。每个变量只能被模拟删除一次，模拟删除后标记为灰。

在步骤 C 中，模拟恢复每个紫色变量。恢复是有条件的，当变量的引用计数大于 0 时才对其做模拟恢复。同样每个变量只能恢复一次，恢复后标记为黑，基本就是步骤 B 的逆运算。这样剩下的一堆没能恢复的就是该删除的蓝色节点了，在步骤 D 中遍历出来真的删除掉。

算法中都是模拟删除、模拟恢复、物理删除，都使用简单的遍历即可（最典型的深度搜索遍历），复杂度为执行模拟操作的节点数正相关，不只是紫色的那些疑似垃圾变量。

默认情况下，PHP 的垃圾回收机制是打开的，而且允许在 php.ini 设置和修改 `zend.enable_gc` 配置项。

当垃圾回收机制打开时，每当根缓存区存满时，就会执行上面描述的循环查找算法。

根缓存区有固定的大小，可以存储 10,000 个可能根，也可以通过修改 PHP 源码文件 Zend/zend_gc.c 中的常量 GC_ROOT_BUFFER_MAX_ENTRIES 并重新编译 PHP 来修改这个 10,000 值。

当垃圾回收机制关闭时，循环查找算法永不执行，实际上可能根就将一直存在根缓冲区中，不管在配置中垃圾回收机制是否激活。

当垃圾回收机制关闭时，如果根缓冲区存满了可能根，更多的可能根显然不会被记录。那些没被记录的可能根，将不会被这个算法来分析处理。如果它们是循环引用周期的一部分，将永不能被清除进而导致内存泄漏。

即使在垃圾回收机制不可用时，可能根也被记录的原因在于相对于每次找到可能根后检查垃圾回收机制是否打开而言，记录可能根的操作更快。不过垃圾回收和分析机制本身要耗不少时间。

除了修改配置 `zend.enable_gc`，也能通过分别调用 gc_enable() 和 gc_disable() 函数来打开和关闭垃圾回收机制。

调用 gc_enable() 和 gc_disable() 函数与修改配置项来打开或关闭垃圾回收机制的效果是一样的。即使在可能根缓冲区还没满时，也能强制执行周期回收，通过调用 gc_collect_cycles() 函数可以达到这个目的，这个函数将返回使用这个算法回收的周期数。

允许打开和关闭垃圾回收机制并且允许自主的初始化的原因，是由于应用程序的某部分可能是高时效性的。在这种情况下，用户就可能不想使用垃圾回收机制。

对应用程序的某部分关闭垃圾回收机制是在冒着可能内存泄漏的风险，因为一些可能根也许存不进有限的根缓冲区，因此就在调用 gc_disable() 函数释放内存之前，先调用

gc_collect_cycles() 函数可能比较明智。因为这将清除已存放在根缓冲区中的所有可能根，然后在垃圾回收机制被关闭时，可留下空缓冲区以有更多空间来存储可能根。

1.5.2 Collecting Cycles

虽然回收可能根有细微的性能上影响，例如记录可能根相对于完全不记录可能根要慢些，不过后来对 PHP run-time 的其他修改减少了这个性能损失。

在进行垃圾回收时产生的性能影响中，第一个是内存占用空间的节省，另一个是垃圾回收机制执行内存清理时的执行时间增加 (run-time delay)。

首先，实现垃圾回收机制的整个原因是为了，一旦先决条件满足，通过清理循环引用的变量来节省内存占用。

在 PHP 执行中，一旦根缓冲区满了或者调用 gc_collect_cycles() 函数时，就会执行垃圾回收。例如，下面的示例显示了在 PHP 5.2 和 PHP 5.3 环境下的内存占用情况，其中排除了脚本启动时 PHP 本身占用的基本内存。

Example 15 GC 内存使用情况

```
<?php
class Foo
{
    public $var = '3.1415962654';
}

$baseMemory = memory_get_usage();

for ( $i = 0; $i <= 100000; $i++ )
{
    $a = new Foo;
    $a->self = $a;
    if ( $i % 500 === 0 )
    {
        echo sprintf( '%8d: ', $i ), memory_get_usage() - $baseMemory, "\n";
    }
}
```

在这个示例中，首先创建了一个对象，这个对象中的一个属性被设置为指回对象本身。在循环的下一个重复 (iteration) 中，当脚本中的变量被重新复制时，就会发生典型性的内存泄漏。

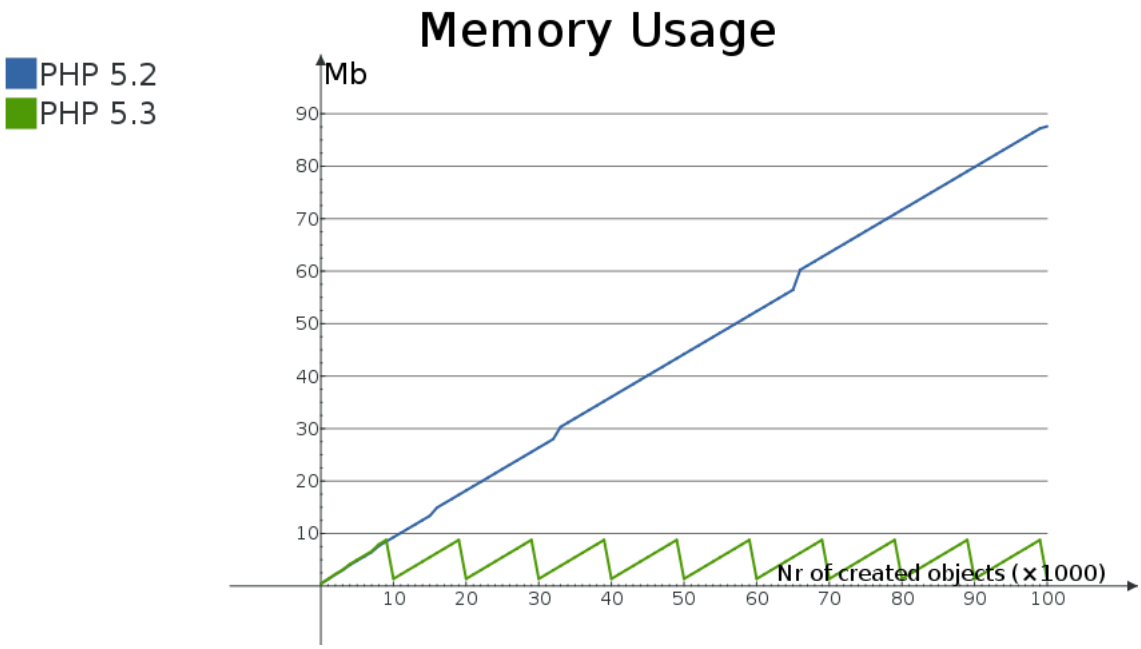

图 1.6: GC 内存使用情况

在这个例子中，两个变量容器是泄漏的 (对象容器和属性容器)，但是仅仅能找到一个可能根——就是被 unset 的那个变量。在 10,000 次重复后 (也就产生总共 10,000 个可能根)，当根缓冲区满时，就执行垃圾回收机制，并且释放那些关联的可能根的内存。

从 PHP 5.3 的锯齿型内存占用图中很容易就能看到，每次执行完 10,000 次重复后，执行垃圾回收，并释放相关的重复使用的引用变量。这里由于泄漏的数据结构非常简单，所以垃圾回收机制本身不必做太多工作。从这个图表中，可以能看到 PHP 5.3 的最大内存占用大概是 9 Mb，而 PHP 5.2 的内存占用一直增加。

垃圾回收影响性能的第二个领域是它释放已泄漏的内存耗费的时间，可以修改上述的示例来查看这个耗时的多少，有更多次数的重复并且删除了循环中的内存占用计算。

Example 16 GC 性能影响

```
<?php
class Foo
{
    public $var = '3.1415962654';
}

for ( $i = 0; $i <= 1000000; $i++ )
{
```

```
    $a = new Foo;
    $a->self = $a;
}
echo memory_get_peak_usage(), "\n";
```

下面运行这个脚本两次，一次通过配置 zend.enable_gc 打开垃圾回收机制时，另一次是它关闭时。

```
$ time php -dzend.enable_gc=0 -dmemory_limit=-1 -n example2.php
10.7s
$ time php -dzend.enable_gc=1 -dmemory_limit=-1 -n example2.php
11.4s
```

后面一次执行比第一次执行的结果在时间上增加了 7%，不过执行这个脚本时内存占用的峰值降低了 98%（从 931Mb 降到 10Mb），从而可以显示出垃圾回收机制在内存占用方面的好处就是在执行中出现更多的循环引用变量时，内存节省的更多的情况下，每次时间增加的百分比都是 7%。

为了在 PHP 内部可以显示更多的关于垃圾回收机制如何运行的信息，需要先重新编译 PHP 使 benchmark 和 data-collecting code 可用，在执行./configure 之前先把环境变量 CFLAGS 设置成-DGC_BENCH=1。

Example 17　启用 GC benchmarking

```
$ export CFLAGS=-DGC_BENCH=1
$ ./config.nice
$ make clean
$ make
```

在打开 PHP 的 benchmark 和 data-collecting code 来重新执行上面的例子代码时，PHP 执行结束后可以看到下面的 GC 统计信息：

```
GC Statistics
-------------
Runs:            110
Collected:       2072204
Root buffer length: 0
Root buffer peak: 10000

     Possible         Remove from Marked
       Root   Buffered    buffer    grey
     -------- -------- ----------- ------
```

```
ZVAL  7175487  1491291   1241690  3611871
ZOBJ 28506264  1527980    677581  1025731
```

其中，主要的信息统计在第一个块，可以看到垃圾回收机制运行了 110 次，而且在这 110 次运行中，总共有超过两百万的内存分配被释放。只要垃圾回收机制运行了至少一次，根缓冲区峰值 (Root buffer peak) 总是 10000.

通常情况下，PHP 中的垃圾回收机制仅仅在循环回收算法确实运行时会有时间消耗上的增加，但是在平常的 (更小的) 脚本中没有性能影响。

在普通的 PHP 脚本中有循环回收机制运行的情况下，内存的节省将允许更多的 PHP 脚本同时运行在服务器上，而且总共使用的内存并不会达到上限。

这种好处在长时间运行脚本（例如长时间的测试套件或者 daemon 脚本等）中尤其明显，同时对通常比 Web 脚本运行时间长的 PHP-GTK 应用程序，新的垃圾回收机制也可以大大改变一直以来认为内存泄漏问题难以解决的看法。

1.6 Development Server

PHP CLI SAPI 提供了一个内置的 Web 服务器，从命令行终端进入项目的 Web 根目录并执行下面的命令就可以启动内置的 Web 服务器来进行本地开发调试。

```
$ cd ~/public_html
$ php -S localhost:8000
PHP Development Server started at Tue Dec 6 10:03:32 2016
Listening on http://localhost:8000
Document root is /home/me/public_html
Press Ctrl-C to quit.
```

URI 请求会被发送到 PHP 所在的的工作目录（Working Directory）进行处理，除非使用了`-t` 参数来自定义不同的目录。

```
$ cd ~/public_html
$ php -S localhost:8000 -t foo/
PHP Development Server started at Tue Dec 6 10:03:32 2016
Listening on http://localhost:8000
Document root is /home/me/public_html/foo
Press Ctrl-C to quit.
```

如果请求未指定执行哪个 PHP 文件，则默认执行目录内的 index.php 或者 index.html。如果这两个文件都不存在，服务器会返回 404 错误。例如，访问 http://localhost:8000/和 http://localhost:8000/myscript.html 的输出窗口如下显示：

```
$ cd ~/public_html
$ php -S localhost:8000
PHP Development Server started at Tue Dec 6 10:03:32 2016
Listening on http://localhost:8000
Document root is /home/me/public_html
Press Ctrl-C to quit.
::1:39144 GET /favicon.ico - Request read
::1:39146 GET / - Request read
::1:39147 GET /favicon.ico - Request read
::1:39148 GET /myscript.html - Request read
::1:39149 GET /favicon.ico - Request read
```

修改 php.ini 中的 cli_server.color 项可以控制内置 Web Server 的终端输出有无颜色。

表 1.2: CLI SAPI cli_server.color 配置选项

名字	默认	可修改范围
cli_server.color	"0"	PHP_INI_ALL

1.6.1 Router

如果在启动 PHP 内置的 Web 服务器时指定了一个 PHP 文件，则这个文件会作为一个“路由”脚本，意味着每次请求都会先执行这个脚本。

```
<?php
// router.php
if (preg_match('/\.(?:png|jpg|jpeg|gif)$/', $_SERVER["REQUEST_URI"]))
    return false; // 直接返回请求的文件
else {
    echo "<p>Welcome to PHP</p>";
}
?>
```

如果指定了路由脚本，那么可以设置请求图片直接显示图片，请求 HTML 则显示“Welcome to PHP”。例如，执行之后终端显示：

```
$ cd ~/public_html
$ php -S localhost:8000 router.php
PHP Development Server started at Tue Dec 6 10:03:32 2016
Listening on http://localhost:8000
Document root is /home/me/public_html
Press Ctrl-C to quit.
```

```
::1:55801 GET /mylogo.jpg - Request read
::1:55803 GET /abc.html - Request read
::1:55804 GET /favicon.ico - Request read
```

如果路由脚本返回 FALSE，那么直接返回请求的文件（例如请求静态文件不作任何处理），否则会把输出返回到浏览器。

1.6.2 Template

模板提供了一种简便的方式，将展现逻辑从控制器和业务逻辑中分离出来。

一般来说，模板包含应用程序的 HTML 代码，但是也可以使用其他的格式（例如 XML）。

模板通常也被称为「视图」，模板其实只是 MVC（模型-视图-控制器）软件架构模式第二个元素的一部份。

模板的主要好处是可以将呈现逻辑与应用程序的其他部分进行分离，模板的单一职责只是呈现格式化后的内容。

模板不负责数据的查询、保存或是其他复杂的任务，从而进一步促成了更干净、更具可读性的代码，开发者在团队协作开发中可以专注服务端的代码（控制器、模型），而设计师则专注客户端代码 (网页) 。

模板还改善了前端代码的组织架构。一般来说，模板放置在「视图」文件夹中，每一个模板都放在独立的一个文件中。

模板的组织方式鼓励代码重用，它将大块的代码拆成较小的、可重用的片段（通常称为局部模板）。举例来说，网站的头、尾区块可以各自定义为一个模板，之后将它们放在每一个页面模板的上、下位置。

最后，根据用户选择的类库，模板可以通过自动转义用户的内容来提供更多的安全性。有些类库可以提供沙箱机制，模板设计者只能使用在白名单中的变量和函数。

1.7 Closure/Anonymous

PHP 从 5.3（2009 年）之后开始引入对闭包以及匿名函数的支持，并在接下来的 PHP 5.4 中增加了将闭包绑定到对象作用域中的特性，并改善其可调用性，从此在大部分情况下可以使用匿名函数取代一般的函数。

1.8 Meta Programming

PHP 通过反射 API 和魔术方法可以实现多种方式的元编程。开发者通过魔术方法（例如 __get()、__set()、__clone()、__toString()、__invoke() 等）可以改变类的行为。

PHP 通过 __call() 和 __callStatic() 可以实现和 Ruby 的 method_missing 方法完成相同的功能。

1.9 Functional Program

函数式编程是一种不同于对象式编程的思想，PHP 并不是天生就属于函数式编程的语言。

PHP 本身的语法、语义不能完全很好地支持函数式编程，不过可以通过封装类库来完成 PHP 对函数式编程的过滤、映射和归约的支持。

```
<?php
/**
 * 函数式编程
 *
 * @author dogstar 20161113
 */


class Functional_Lite {

    protected $data;

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


    /** ------------ middle operation ------------ **/

    public function filter($callback) {
        $newData = array();

        if (!is_callable($callback)) {
            $callback = create_function('$it', 'return ' . $callback . ';');
        }
```

```
        foreach ($this->data as $item) {
            if (call_user_func($callback, $item)) {
                $newData[] = $item;
            }
        }

        $this->data = $newData;

        return $this;
    }

    public function map($callback) {
        if (!is_callable($callback)) {
            $callback = create_function('$it', 'return ' . $callback . ';');
        }

        foreach ($this->data as &$itemRef) {
            $itemRef = call_user_func($callback, $itemRef);
        }

        return $this;
    }

    /** ------------ end operation ------------ **/

    public function reduce($callback, $accumulator) {
        $rs = $accumulator;

        if (!is_callable($callback)) {
            $callback = create_function('$it_1, $it_2', 'return ' . $callback . ';');
        }

        foreach ($this->data as $item) {
            $rs = call_user_func_array($callback, array($rs, $item));
        }

        return $rs;
    }


    public function foldLeft($callback, $accumulator = NULL) {
```

```
        // default
        if ($accumulator === NULL) {
            $accumulator = array_shift($this->data);
        }

        return $this->reduce($callback, $accumulator);
    }

    public function foldRight($callback, $accumulator = NULL) {
        $this->data = array_reverse($this->data);
        return $this->foldLeft($callback, $accumulator);
    }

    /** ------------ extra operation ------------ **/

}
```

假设我们需要计算 1 到 5 之间的偶数之和的两倍，那么：

```
$fun = new Functional_Lite(range(1, 5));


$rs = $fun->filter(function ($x) { return $x % 2 == 0;})
          ->map(function ($x) {return $x * 2;})
          ->reduce(function ($x, $y) { return $x + $y;}, 0);

var_dump($rs); // 2 * 2 + 4 * 2 = 12
```

使用字符串表达式进一步缩短代码，可以这样：

```
$fun = new Functional_Lite(range(1, 5));


$rs = $fun->filter('$it % 2 == 0')
          ->map('$it * 2')
          ->reduce('$it_1 + $it_2', 0);

var_dump($rs); // 2 * 2 + 4 * 2 = 12
```

这里使用了 $it 来表示每次迭代的元素。例如，对于 reduce 操作，使用了 $it_1、$it_2 分别表示第一个元素、第二个元素。

除了对数值进行操作，也可以对字符串或其他类型的集合进行操作。假设需要过滤一个字符的单词，然后把剩下的单词转大写，再用空格拼接起来。

```
$fun = new Functional_Lite(array('dogstar', 'x', 'love', 'y', 'yoyo', '!!!'));

$rs = $fun->filter(function ($x) { return strlen($x) > 1;})
          ->map(function ($x) { return strtoupper($x);})
          ->reduce(function ($x, $y) { return $x . ' ' . $y;}, 'RS:');

var_dump($rs); // RS: DOGSTAR LOVE YOYO !!
```

对于最后的结果，我们在累加量初始了“RS:”，所以最后的结果里有“RS:”的前缀。

同样地，使用字符串表达式可以简化上面的处理以达到同样的效果：

```
$rs = $fun->filter('strlen($it) > 1')
          ->map('strtoupper($it)')
          ->reduce('$it_1 . " " . $it_2', 'RS:');
```

PHP 实际上距离函数式编程的道路还很遥远，比如柯里化、部分施用、偏函数、Either 类、Opeion 类、值不可变，而且有很多在 PHP 中是明显很难实现的（例如上下文的处理）。

1.10 Dependency Injection

依赖注入是一种允许用户从硬编码的依赖中解耦出来，从而在运行时或者编译时能够修改的软件设计模式。

具体来说，依赖注入通过构造注入、函数调用或者属性的设置来提供组件的依赖关系。

下面我们可以用一个简单的例子来说明依赖注入的概念，代码中有一个 Database 的类需要一个适配器来与数据库交互，在构造函数里实例化了适配器就会产生耦合，使得测试变得很困难，而且 Database 类和适配器耦合的很紧密。

```
<?php
namespace Database;

class Database
{
    protected $adapter;

    public function __construct()
    {
        $this->adapter = new MySqlAdapter;
    }
}
```

```
class MysqlAdapter {
}
```

上述这段代码可以用依赖注入进行重构来实现解耦。

```
<?php
namespace Database;


class Database
{
   protected $adapter;

   public function __construct(MySqlAdapter $adapter)
   {
      $this->adapter = $adapter;
   }
}

class MySqlAdapter {
}
```

依赖注入使得代码通过外界给予 Database 类的依赖，而不是让它自己产生依赖的对象，还可以使用可以接受依赖对象参数的成员函数来设置，或者如果 $adapter 属性本身是 public 的就可以直接给它赋值。

“控制反转”(Inversion of Control) 或者“依赖反转准则”(Dependency Inversion Principle) 是依赖注入可以解决的更复杂的问题。

1.10.1 Inversion of Control

顾名思义，一个系统通过组织控制和对象的完全分离来实现”控制反转”。对于依赖注入，这就意味着通过在系统的其他地方控制和实例化依赖对象来实现解耦。

虽然 PHP 框架可以实现控制反转，不过现实问题是应该反转哪些部分以及到反转到什么程度。比如，MVC 框架通常会提供超类（Superclass）或者基本的控制器类以便其他控制器可以通过继承来获得相应的依赖，这就是控制反转的例子，但是这种方法是直接移除了依赖而不是减轻了依赖。

依赖注入允许通过按需注入的方式更加优雅地解决上述这个问题，完全不需要任何耦合。

1.10.2 Dependency Inversion

依赖反转准则（Dependency Inversion Principle）是面向对象设计准则 S.O.L.I.D 中的“D”，倡导“依赖于抽象而不是具体”。简单来说，依赖反转准则就是指依赖应该是接口/约定或者抽象类，而不是具体的实现。

接下来开始重构前面的例子使其遵循依赖反转准则：

```
<?php
namespace Database;

class Database
{
    protected $adapter;

    public function __construct(AdapterInterface $adapter)
    {
        $this->adapter = $adapter;
    }
}

interface AdapterInterface {}

class MysqlAdapter implements AdapterInterface {}
```

遵循依赖反转准则的实现（例如 Database 类）依赖于接口，相比依赖于具体实现有更多的优势。例如，假设工作团队中的一位同事负责设计适配器，那么在第一个例子中，我们需要等待适配器设计完之后才能单元测试。现在由于依赖是一个接口/约定，于是我们能轻松地模拟接口测试，因为我们知道同事会基于约定实现那个适配器

依赖反转的一个更大的好处是代码扩展性变得更高。例如，如果一段时间之后我们决定要迁移到一种不同的数据库，接下来只需要写一个实现相应接口的适配器并且注入进去，由于适配器遵循接口的约定，因此不再需要额外的重构。

1.11 Framework Container

依赖注入容器和依赖注入不是相同的概念。容器是帮助用户更方便地实现依赖注入的工具，但是它们通常被误用来实现反模式设计 Service Location 。

把一个依赖注入容器作为 Service Locator 注入进类中隐式地建立了对于容器的依赖，而不是真正需要替换的依赖，而且还会让代码更不透明，最终变得更难测试。

大多数现代的框架都有自己的依赖注入容器，允许通过配置将依赖绑定在一起，这样就可以写出和框架层同样干净、解耦的应用层代码。

1.12 Commandline Interface

PHP 提供的 CLI SAPI（Server Application Programming Interface，服务端应用编程端口）允许使用 PHP 开发命令行应用。

- CLI SAPI 和其它 CLI SAPI 模块相比有很多的不同之处；
- CLI 和 CGI 是不同的 SAPI，尽管它们之间有很多共同的行为。

PHP 的命令行（CLI）脚本接口可以帮助用户完成自动化的任务（例如测试、部署和应用管理），而且可以直接调用程序代码而无需创建 Web 图形界面。

用户执行 `php -a` 选项就可以进入和 Ruby 的 IRB 或 python 的交互式 shell 相似的交互式 shell。例如，一个简单的“Hello, $name”CLI 程序：

```
<?php
if ($argc != 2) {
    echo "Usage: php hello.php [name].\n";
    exit(1);
}
$name = $argv[1];
echo "Hello, $name\n";
```

PHP 会在脚本运行时根据参数设置两个特殊的变量，$argc 是一个表示参数个数的整数，$argv 是一个数组变量，包含每个参数的值，它的第一个元素一直是 PHP 脚本的名称（例如这里就是 hello.php）。

```
$ php -a
Interactive mode enabled

> php hello.php
Usage: php hello.php [name]
> php hello.php world
Hello, world
```

命令运行失败时，可以通过 exit() 表达式返回一个非 0 整数来通知 shell。

以下为 CLI SAPI 和其它 CLI SAPI 模块相比的显著区别：

1. 与 CGI SAPI 不同，其输出没有任何头信息。
   尽管 CGI SAPI 提供了取消 HTTP 头信息的方法，但在 CLI SAPI 中并不存在类似的方法以开启 HTTP 头信息的输出。

CLI 默认以安静模式开始，但为了保证兼容性，`-q` 和`--no-header` 参数为了向后兼容仍然保留，使得可以使用旧的 CGI 脚本。

在运行时，PHP CLI 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)。

出错时输出纯文本的错误信息（非 HTML 格式)。

2. CLI SAPI 强制覆盖了 php.ini 中的某些设置，这些设置在命令行环境下是没有意义的。

   CLI SAPI 覆盖的 php.ini 设置选项包括 html_errors，implicit_flush，max_execution_time 和 register_argc_argv。

3. CLI SAPI 不会将当前目录改为已运行的脚本所在的目录。

   可以在命令行运行时给该 CGI SAPI 加上`-C` 参数，使其支持 CLI SAPI 的功能。

表 1.3: CLI SAPI 强制覆盖的设置选项

设置选项	CLI SAPI 默认值	备注
html_errors	FALSE	无意义的 HTML 标记符使出错信息很凌乱，所以在命令行下阅读报错信息十分困难，因此将该选项的默认值改为 FALSE。
implicit_flush	TRUE	命令行模式所有来自 print 和 echo 的输出将被立即写到输出端，而不作任何地缓冲操作。如果希望延缓或控制标准输出，仍然可以使用 output buffering 设置项。
max_execution_time	0（无限值）	命令行环境下使用 PHP 时取消了最大运行时间的限制，命令行应用程序可以在后台持久运行。
register_argc_argv	TRUE	设置为 TRUE 就总是可以在 CLI SAPI 中访问到 argc (传送给应用程序参数的个数) 和 argv（包含有实际参数的数组)。

上述这些设置无法在 php.ini 或任何指定的其它文件中被初始化为其它值，这些默认值被限制在所有其它的设置文件被解析后改变，不过这些设置的值可以在程序运行的过程中被改变（尽管对于该运行过程来说，这些设置项是没有意义的)。

CLI/CGI 二进制执行文件的文件名、位置和是否存在会根据 PHP 在系统上的安装而不同，在命令行下运行 `php -v` 就可以输出该 php 是 CGI 还是 CLI 的相关信息。

在默认情况下，当运行 make 时，CGI 和 CLI 都会被编译并且分别放置在 PHP 源文件目录的 sapi/cgi/php 和 sapi/cli/php 下，而且这两个文件都被命名为 php。

在 make install 的过程中会发生的行为取决于配置行。

- 如果在配置的时候选择了一个 SAPI 模块（例如 apxs），或者使用了`--disable-cgi`参数，则在 make install 的过程中，CLI 将被拷贝到 {PREFIX}/bin/php，除非 CGI 已经被放置在了那个位置。
- 如果希望撤销 CGI 执行文件的安装，需要在 make install 之后运行 make install-cli，或者在配置行中加上`--disable-cgi`参数。

`--enable-cli`和`--enable-cgi`同时默认有效，因此不需要再配置行中加上`--enable-cli`来使得 CLI 在 make install 过程中被拷贝到 {PREFIX}/bin/php。

CLI SAPI 提供了命令行环境下专用的常量，例如 STDIN、STDOUT 和 STDERR，这样用户就无需自己建立指向诸如 stderr 的流，只需简单的使用这些常量来代替流指向即可，同时无需用户自己来关闭这些流，PHP 会自动完成这些操作。

```
$ php -r 'fwrite(STDERR, "stderr\n");'
```

- STDIN 是一个已打开的指向 stdin 的流。可以用如下方法来调用：

  ```
  <?php
  $stdin = fopen('php://stdin', 'r');
  ```

- STDOUT 是一个已打开的指向 stdout 的流。可以用如下方式来调用：

  ```
  <?php
  $stdout = fopen('php://stdout', 'w');
  ```

- STDERR 是一个已打开的指向 stderr 的流。可以用如下方式来调用：

  ```
  <?php
  $stderr = fopen('php://stderr', 'w');
  ```

如果需要从 stdin 读取一行内容，可以使用如下方式：

```
<?php
$line = trim(fgets(STDIN)); // 从 STDIN 读取一行
fscanf(STDIN, "%d\n", $number); // 从 STDIN 读取数字
```

1.12.1 $argv

除了 STDIN、STDOUT 和 STDERR 之外，PHP CLI 没有限制传送给脚本程序的参数的个数（shell 程序对命令行的字符数有限制，但通常都不会超过该限制）。

命令行中传递给脚本的参数可在全局变量 $argv 中获取。该数组中下标为零的成员为脚本的名称（当 PHP 代码来自标准输入获直接用 -r 参数以命令行方式运行时，该名称为“-”）。

1.12.2 $argv

全局变量 $argc 存有 $argv 数组中成员变量的个数（而非传送给脚本程序的参数的个数）。

向脚本传送以 - 开头的参数会导致错误，因为 PHP 会认为应该由它自身来处理这些参数。可以用参数列表分隔符--来解决这个问题。在 PHP 解析完参数后，该符号后所有的参数将会被原样传送给脚本程序。

- 以下命令将不会运行 PHP 代码，而只显示 PHP 命令行模式的使用说明：

```
$ php -r 'var_dump($argv);' -h
Usage: php [options] [-f] <file> [args...]
[...]
```

- 以下命令将会把“-h”参数传送给脚本程序，PHP 不会显示命令行模式的使用说明：

```
$ php -r 'var_dump($argv);' -- -h
array(2) {
  [0]=>
  string(1) "-"
  [1]=>
  string(2) "-h"
}
```

1.12.3 shibang

PHP 的命令行模式能使得 PHP 脚本能完全独立于 web 服务器单独运行。

- 如果使用 Unix 系统，需要在 PHP 脚本的最前面加上一行特殊的代码，使得它能够被执行，这样系统就能知道用哪个程序去运行该脚本。
- 如果使用 Windows 系统，可以将 php.exe 和.php 文件的双击属性相关联，也可以编写一个批处理文件来用 PHP 执行脚本。

为 Unix 系统增加的第一行代码不会影响该脚本在 Windows 下的运行，因此也可以用该方法编写跨平台的脚本程序。具体来说，在 PHP 脚本的第一行以 `#!/usr/bin/php` 开头，在其后加上以 PHP 开始和结尾标记符包含的正常的 PHP 代码，然后为该文件设置正确的运行属性（例如`-r-xr-xr-x`），就可以使得该文件能够像 shell 脚本或 PERL 脚本一样被直接执行。

```
#!/usr/bin/php
<?php
var_dump($argv);
```

在向上述该脚本传送以 - 开头的参数时，脚本仍然能够正常运行。

```
$ chmod +x test
$ ./test -h -- foo
array(4) {
  [0]=>
  string(6) "./test"
  [1]=>
  string(2) "-h"
  [2]=>
  string(2) "--"
  [3]=>
  string(3) "foo"
}
```

下面是一个以命令行方式运行的 PHP 脚本的示例。

```
#!/usr/bin/php
<?php
if ($argc != 2 || in_array($argv[1], array('--help', '-help', '-h', '-?'))) {
?>

This is a command line PHP script with one option.

  Usage:
  <?php echo $argv[0]; ?> <option>

  <option> can be some word you would like
  to print out. With the --help, -help, -h,
  or -? options, you can get this help.

<?php
} else {
    echo $argv[1];
}
?>
```

这个示例程序检查了参数的个数是大于 1 个还是小于 1 个，而且如果参数是`--help`，`-help`，`-h` 或 `-?` 时，打印出帮助信息，并同时动态输出脚本的名称，最后如果还收到了其它参数则显示其内容。

在第一行中用特殊的代码来指明该文件应该由 PHP 来执行。在这里使用 CLI 的版本，因此不会有 HTTP 头信息输出。

在用 PHP 编写命令行应用程序时，可以使用两个参数 $argc 和 $argv。其中，前面一个的值是比参数个数大 1 的整数（运行的脚本本身的名称也被当作一个参数），第二个是包含有参数的数组，其第一个元素为脚本的名称，下标为数字 0（$argv[0]）。

如果希望在 Unix 下运行以上脚本，需要使其属性为可执行文件，然后简单的运行 script.php echothis 或 script.php -h。在 Windows 下，可以为此编写一个批处理文件：

```
@C:\php\php.exe script.php %1 %2 %3 %
```

上述这个批处理文件会帮助将附加的参数传给脚本程序：`script.bat echothis` 或 `script.bat -h`。

Chapter 2

Syntax

2.1 Constructs

PHP 的语法和语义说明了 PHP 代码中的文本和符号的意义，它们定义了 PHP 程序如何编写和解析，以及 PHP 代码如何嵌入到 HTML 文档中。

= 用于变量赋值操作，每条 PHP 语句以逗号结束，可以使用 echo 语句来输出 PHP 变量（以 $ 开头）的内容，同时 PHP 命名是大小写敏感的。

用户可以使用 function 和 class 来进行代码模块化，而且 PHP 支持 if、while、for、foreach、switch 等控制结构，同时 PHP 也支持类似 end* 的控制代码结构。

```
if ($x==0) :
  echo "zero";
endif;
```

PHP 原生的模板引擎功能支持把 PHP 代码直接嵌入到 HTML 中，例如：

```
<!DOCTYPE html>
<html>
<meta charset="utf-8">
<head>
  <title>PHP Test</title>
</head>
<body>
  <?php
   echo 'Hello World';
  ?>
</body>
</html>
```

2.2 Delimiters

PHP 引擎根据 `<?php` 和`?>` 分隔符来解析 PHP 代码。

PHP 会在输出时自动删除其结束符`?>` 后的一个换行，主要是针对在一个页面中嵌入多段 PHP 代码或者包含了无实质性输出的 PHP 文件而设计。如果确实需要在 PHP 结束符`?>` 之后输出换行，可以在其后加一个空格，或者在最后的一个 `echo/print` 语句中加入一个换行。

2.3 Variables

PHP 中，变量以“$”后接变量名称来表示，而且变量名称区分大小写。

- 变量是程序运行时用于保存信息或数据的临时“仓库”，可以改变或删除变量。
- 变量和常量可以理解为程序中的基本数据存储单元，可以用来存储不同类型的数据。
- PHP 的变量和常量的数据类型由程序的上下文决定。

有效的变量名称应以字母或下划线开头，后可以接任意数目的字母、数字或下划线，PHP 也支持使用多字节文字作为变量名。

PHP 解释器在解析变量时，将会尽可能多地取得“$”后面的字符来组成一个合法的变量名，可以使用花括号把变量名括起来以明确表示一个变量。

2.4 Structures

PHP 使用 `{...}` 来放置控制结构中的代码，或者也可以使用类似如下的结构来进行逻辑控制。

```
if (condition) :
    // code here
elseif (condition) :
    // code here
else :
    // code here
endif;
```

PHP 的语法参考了 Perl、C 语言，而且可以内嵌在 HTML 之中。

```
<!DOCTYPE html>
<meta charset="utf-8">
<html>
```

```
<head>
 <title>PHP Test</title>
</head>
<body>
 <?php
  echo 'Hello World';
 ?>
</body>
</html>
```

PHP 解析引擎只解析 <?php 到?> 之间的代码，不包含在 <?php 到?> 之间的内容会直接输出。或者说，<?php 和?> 可以让源代码不断地进出 PHP 模式，并最终生成解析结果。

```
<?php
  //-PHP代码
?>
html内容
<?php
  //-PHP代码
?>
```

在 HTML 中嵌入 PHP 时，比如需要单独输出某个变量，除了正常采用 echo 语句外，可以直接采用<?=$title ?>

在判断语句中的 HTML 代码并不会被直接输出，而且根据 PHP 代码的执行来决定，例如：

```
<?php
 if (false) {
?>
HTML Code
<?php
 }
?>
```

PHP 可以用三种注解的形式：C 与 C++ 所使用的“/*...*/”与“//”，和 Perl 的“#”。

```
<?php
/* 多行注释的第一行
多行注释的另一行 */

// 单行注释
```

```
# 单行注释
?>
```

下面的示例说明了 PHP 的变量、常量、控制流程、函数、程序标记和注释等基本元素。

```
<html>
<head><title>Test</title></head>
<body>
<?php
define("MAX_LINE_NUM",4);
$title="<h1>Hello,php</h1>\n";
echo $title;

echo "<pre>\n";
for($i=1;$i<MAX_LINE_NUM;$i++){
    echo print_star($i);
    echo "\n";
}
echo "</pre>\n";

function print_star($num){
    return str_repeat("*",$num);
}

?>
</body>
</html>
```

在对函数或类的定义进行注释说明时，应该包含功能说明、参数列表和返回值等信息，也可以加入应用范例。

Chapter 3

Encoding

PHP 没有在底层实现对 Unicode 的支持。虽然有很多途径可以确保 UTF-8 字符串能够被正确地处理，但是这并不是很简单，通常需要对 Web 应用进行全方面的检查（从 HTML 到 SQL 再到 PHP）。

3.1 String

最基本的字符串操作（例如连结两个字符串或将字符串赋值给变量）往往并不需要对 UTF-8 做特别的处理，然而大多数字符串的函数（例如 strpos() 和 strlen()）确实需要特别的处理。

字符串函数名中通常包含 mb_*（例如 mb_strpos() 和 mb_strlen()），这些 mb_* 字符串是由 mbstring（Multibyte String Extension）提供支持并专门为操作 Unicode 字符串而特别设计。

在操作 Unicode 字符串时务必使用 mb_* 函数。例如，如果对一个 UTF-8 字符串使用 substr()，那返回的结果中有很大可能会包含一些乱码，正确的方式是使用 mb_substr()。

最难的地方在于每次都要记得使用 mb_* 函数，哪怕只有一次忘记了使用，Unicode 字符串就有在接下来的过程中变成乱码的风险。

事实上，不是所有的字符串函数都有一个对应的 mb_* 函数，应该在所有的 PHP 脚本（或全局包含的脚本）的开头使用 mb_internal_encoding() 函数，然后紧接着在可能对浏览器进行输出的脚本中使用 mb_http_output()，在每一个脚本当中明确声明字符串的编码可以避开风险。

另外，许多对字符串进行操作的函数都有一个可选的参数用来指定字符串编码。当可以设定这类参数时，应该始终明确指定使用 UTF-8。例如，htmlentities() 有一个字符编

码的选项，应该始终将其设为 UTF-8[1]。

最后，如果分布式的应用程序中不能确定 mbstring 扩展一定开启的话，可以考虑使用 patchwork/utf8 包，这样就会在 mbstring 可用时自动使用，否则自动切换回非 UTF-8 函数。

3.1.1 strpos()

3.1.2 strlen()

3.1.3 substr()

3.1.4 mb_strpos()

3.1.5 mb_strlen()

3.1.6 mb_substr()

3.2 Browser

3.2.1 htmlentities()

3.2.2 htmlspecialchars()

3.2.3 mb_http_output()

使用 mb_http_output() 函数来确保 PHP 向浏览器输出 UTF-8 格式的字符串。

浏览器需要接收 HTTP 应答来指定页面是由 UTF-8 进行编码的。以前这一步是通过在页面 <head> 标签下包含字符集 <meta> 标签实现的，这是一种可行的方式，现在更好的做法是在 Content-Type 响应头中进行设置，可以让处理速度更快。

```
<?php
// Tell PHP that we're using UTF-8 strings until the end of the script
mb_internal_encoding('UTF-8');

// Tell PHP that we'll be outputting UTF-8 to the browser
mb_http_output('UTF-8');

// Our UTF-8 test string
$string = 'Êl síla erin lû e-govaned vîn.';
```

[1]从 PHP 5.4.0 开始, htmlentities() 和 htmlspecialchars() 的编码都已经被默认设为了 UTF-8。

```
// Transform the string in some way with a multibyte function
// Note how we cut the string at a non-Ascii character for demonstration purposes
$string = mb_substr($string, 0, 15);

// Connect to a database to store the transformed string
// See the PDO example in this document for more information
// Note the `charset=utf8mb4` in the Data Source Name (DSN)
$link = new PDO(
   'mysql:host=your-hostname;dbname=your-db;charset=utf8mb4',
   'your-username',
   'your-password',
   array(
      PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
      PDO::ATTR_PERSISTENT => false
   )
);

// Store our transformed string as UTF-8 in our database
// Your DB and tables are in the utf8mb4 character set and collation, right?
$handle = $link->prepare('insert into ElvishSentences (Id, Body) values (?, ?)');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->bindValue(2, $string);
$handle->execute();

// Retrieve the string we just stored to prove it was stored correctly
$handle = $link->prepare('select * from ElvishSentences where Id = ?');
$handle->bindValue(1, 1, PDO::PARAM_INT);
$handle->execute();

// Store the result into an object that we'll output later in our HTML
$result = $handle->fetchAll(\PDO::FETCH_OBJ);

header('Content-Type: text/html; charset=UTF-8');
?><!doctype html>
<html>
   <head>
      <meta charset="UTF-8">
      <title>UTF-8 test page</title>
   </head>
   <body>
      <?php
```

```
        foreach($result as $row){
            print($row->Body); // This should correctly output our transformed UTF-8
                string to the browser
        }
        ?>
    </body>
</html>
```

3.2.4 mb_internal_encoding()

3.3 Database

如果使用 PHP 来操作 MySQL，即使做到了上面的每一点，字符串仍可能面临在数据库中以非 UTF-8 的格式进行存储的问题。

为了确保字符串从 PHP 到 MySQL 都使用 UTF-8，需要检查确认数据库和数据表都设定为 utf8mb4 字符集和整理，并且确保 PDO 连接请求也使用了 utf8mb4 字符集。

在实践中，为了完整的 UTF-8 支持必须使用 utf8mb4 而不是 utf8。

Chapter 4

Data types

PHP 主要有四种标量类型，标量可以认为是数据结构中最基本的单元，只能存储一个固定的数据。

- 整型（integer）
- 浮点型（float）
- 布尔型（boolean）
- 字符串（string）

在定义字符串数据类型时，单引号和双引号在功能上有明显的区别。

- 双引号字符串支持变量的解析和转义字符。
- 单引号字符串不支持变量解析以及转义字符，如果单引号字符串中包含“'”，那么需要在前面加一个“\”进行转义。

单引号在定义简单的字符串时是更加高效的处理方式，双引号则需要花费额外的开销来处理字符串的转义和变量的解析，因此在没有特别需求时可以使用单引号。

另外，PHP 还支持使用定界符 <<< 来定义字符串，使用上字符串必须包含在一组定界标识符中，而且定界结束符必须顶格书写。

```
<?php
$string=<<<EOD
  Are you ok?
EOD;

echo <<< EOT
  $string
  I'm ok.
EOT;
?>
```

定界符“<<<”后面紧跟的就是定界标识符，同样可以由字母、数字或下划线组成，并且不能以数字开始。

定界符中的字符串可以被解析，可以使用转义字符，但是不必转义引号，定界符可以用于定义或输出大量的文本的情况。

PHP 支持的两种复合类型分别是数组（array）和对象（object），相对于标量只能存放一个数据，数组可以存放任何类型的多个数据，可以是标量数据、数组、对象、资源以及其他语法结构等。

- 数组中的每一个数据称为一个元素，每个元素包括索引（键名）和值两部分。
- 元素的索引只能由数字或字符串组成，使用整型数字表示索引时类似于 C 语言的数组“下标 (index)”。
- 元素的值可以是多种类型。

PHP 提供了多种方法来构造一个数组，其中最简单的是使用数组函数 array()，而且在定义数组时可以省略索引。

```
<?php
$key = 7;
$my_stuff = array(
  1,
  'abc',
  'book'=>'PHP',
  13=>'number',
  "Children's Day"=>'1st June',
  $ky=>'a dynamic day',
  'what is the key here', // 14=>'what is the key here'
);
?>
```

省略的索引值将按当前下标最大索引值加 1 计算，因此如果数组中当前的最大索引值为负数，那么下一个索引值变为 0。

数组允许分别对元素赋值，如果有同名元素，那么只保留最后一次的定义，而且所有整数字符的键都被看作是整数。

4.1 Object

对象是一种高级的数据类型，任何事物都可以看作是对象，每一个对象都是由一组属性值和一组方法（或称操作）构成，其中属性表明对象的一种状态，方法通常用来获取或改变对象的属性。

PHP 支持的两种特殊类型分别是 NULL 和资源类型，其中 NULL 类型表示“空”、“没有”、“不存在”或“未被初始化”等概念，它只有一个值（即 NULL）。

NULL 和 False 可以表达相同的概念，但是二者有着本质的区别。实际上，任何一种类型的数据，如果没有初始化或者被清空，都可以被认为是 NULL。

- 布尔型变量可以转换为整数：True 转化为 1，False 转换为 0。
- 布尔型变量可以转换为字符串：True 转化为字符“1”，False 转换为字符“0”。

NULL 转换为整型后的值为 0，转换为字符串后的值为空字符“”。

4.2 String

PHP 提供了针对字符串和数组的完善的支持，并且 PHP 还提供了自己独特的功能扩展。

4.2.1 curl

PHP 利用 curl 库可以对 URL 地址进行解码或编码，不需要自己手动实现。

4.2.2 pcre

PHP 实用 PCRE 来实现对正则表达式的支持。

4.2.3 pspell

PHP 利用 aspell 库（http://aspell.net）可以支持 pspell 扩展，从而实现单词拼写检查和建议。

4.3 Array

PHP 通过 serialize() 和 unserialize() 等函数，可以将数组、对象等数据进行序列化编码，从而方便其在数据库或会话周期中的存储。

4.3.1 wddx

PHP 利用 WDDX（Web 分布式数据格式）可以进行数据串行化，从而在应用环境中交换更为复杂的数据结构。

4.4 Resource

资源是由专门的函数建立和使用的，资源本身是一种特殊的数据类型，可以被用户创建、使用和释放。

PHP 实现了自动垃圾回收机制来释放不再需要的资源，一般发生在 PHP 脚本运行结束之后。

按照封装的层次，用户无法获知某个资源的细节，通常包含打开文件、数据库连接、图形画布区域等特殊句柄等。

Chapter 5

Functions

PHP 函数提供了各种不同的功能，例如文件处理、FTP、字符串处理等，而且这些函数的使用方法和 C 语言相近（例如 printf）。

PHP 的函数库系统除了提供了大量的内置函数之外，还允许用户通过扩展来引入自己的函数。例如，用户可以通过下面的代码来定义自己的函数 myFunction()。

```
function myFunction() { // declares a function, this is named myFunction
 return 'John Doe'; // returns the value 'John Doe'
}

echo 'My name is ' . myFunction() . '!';
// outputs the text concatenated with the return value of myFunction.
// myFunction is called as a result of this syntax.
// The result of the output will be 'My name is John Doe!'
```

在 Zend 引擎的支持和优化下，现在 PHP 可以通过不同的扩展模块来支持各种各样的 Web 应用（例如网络爬虫等）。

```
<?php
function hello(){
  echo "Hello World!\n";
}

hello();
?>
```

Prior to version 5.3, PHP supports quasi-anonymous functions through the `create_function()` function, although they are not true anonymous functions because anonymous functions are nameless, but functions can only be referenced by name, or indirectly through a variable `$function_name();`,

in PHP. As of version 5.3, PHP supports true anonymous functions.

Function calls may be made via variables, where the value of a variable contains the name of the function to call. This is illustrated in the following example:

```
<?php
function hello(){
  return 'Hello';
}
function world(){
  return "World!";
}

$function1 = 'hello';
$function2 = 'world';

echo $function1() . ' ' . $function2();
?>
```

PHP does not support named parameters or parameter skipping. Some core PHP developers have publicly expressed disappointment with this decision. Others have suggested workarounds for this limitation.

PHP gained support for closures in PHP 5.3. True anonymous functions are supported using the following syntax:

```
function getAdder($x) {
 return function($y) use ($x) {
  return $x + $y;
 };
}

$adder = getAdder(8);
echo $adder(2); // prints "10"
```

Here, the `getAdder()` function creates a closure using the parameter `$x` (the keyword use imports a variable from the lexical context), which takes an additional argument `$y` and returns it to the caller. Such a function is a first class object, meaning that it can be stored in a variable, passed as a parameter to other functions, etc. For more details see Lambda functions and closures RFC.

The goto flow control statement is used as follows:

```
function lock() {
```

```
$file = fopen('file.txt', 'r+');
retry:
if (!flock($file, LOCK_EX | LOCK_NB)) {
 goto retry;
}
fwrite($file, 'Success!');
fclose($file);
}
```

When `flock()` is called, PHP opens a file and tries to lock it. The target label `retry:` defines the point to which execution should return if `flock()` is unsuccessful and goto retry; is called. The `goto` statement is restricted and requires that the target label be in the same file and context.

The goto statement has been supported since PHP 5.3.

Chapter 6

Objects

Basic object-oriented programming functionality was added in PHP 3 and improved in PHP 4. Object handling was completely rewritten for PHP 5, expanding the feature set and enhancing performance.

In previous versions of PHP, objects were handled like value types(or primitive types). The drawback of this method was that the whole object was copied when a variable was assigned or passed as a parameter to a method. In the new approach, objects are referenced by handle, and not by value.

PHP 5 introduced private and protected member variables and methods, along with `abstract classes`, `final classes`, `abstract methods`, and `final methods`. It also introduced a standard way of declaring `constructors` and `destructors`, similar to that of other object-oriented languages such as C++, and a standard exception handling model.

Furthermore, PHP 5 added interfaces and allowed for multiple interfaces to be implemented. There are special interfaces that allow objects to interact with the runtime system. Objects implementing ArrayAccess can be used with array syntax and objects implementing Iterator or IteratorAggregate can be used with the `foreach` language construct.

The static method and class variable features in Zend Engine 2 do not work the way some would expect. There is no virtual table feature in the engine, so static variables are bound with a name instead of a reference at compile time.

This example shows how to define a class, `foo`, that inherits from class `bar`. The function `mystaticfunc` is a public static function that can be called with `foo::mystaticfunc();`.

```
class foo extends bar{
  function __construct(){
```

```
  $doo = "wah dee dee";
 }
 public static function mystaticfunc(){
  $dee = "dee dee dum";
 }
}
```

If the developer creates a copy of an object using the reserved word `clone`, the Zend engine will check if a `__clone()` method has been defined or not. If not, it will call a default `__clone()` which will copy the object's properties. If a `__clone()` method is defined, then it will be responsible for setting the necessary properties in the created object. For convenience, the engine will supply a function that imports the properties of the source object, so that the programmer can start with a by-value replica of the source object and only override properties that need to be changed.

The following is a basic example of object-oriented programming in PHP:

```
<!DOCTYPE html>
<html>
 <head>
  <title>testobject</title>
 </head>
 <body>
  <?php
   class Person{
    public $firstName;
    public $lastName;

    public function __construct($firstName,$lastName= ''){
     $this->firstName = $firstName;
     $this->lastName = $lastName;
    }

    public function greet(){
     return "Hello, my name is " . $this->firstName . " " . $this->lastName . ".";
    }

    public static function staticGreet($firstName,$lastName){
     return "Hello, my name is " . $firstName . " " . $lastName . ".";
    }
   }
```

```
    $he = new Person('Jim','Green');
    $she = new Person('Meimei','Han');
    $other = new Person('LiPing');

    echo $he->greet();
    echo '<br />';
    echo $she->greet();
    echo '<br />';
    echo $other->greet();
    echo '<br />';
    echo Person::staticGreet('LiPing');
  ?>
 </body>
</html>
```

The visibility of PHP properties and methods is defined using the keywords `public`, `private`, and `protected`. The default is public, if only `var` is used; `var` is a synonym for public. Items declared `public` can be accessed everywhere. `protected` limits access to inherited classes (and to the class that defines the item). `private` limits visibility only to the class that defines the item. Objects of the same type have access to each other's private and protected members even though they are not the same instance. PHP's member visibility features have sometimes been described as "highly useful." However, they have also sometimes been described as "at best irrelevant and at worst positively harmful."

Chapter 7

Filesystem

文件系统是操作系统的重要组成部分，可以将其理解为数据存储的基本单元。

7.1 Directory

PHP 提供的文件和目录函数等可以对本地文件系统进行直接操作，例如创建、读取和删除文件或目录，也可以对文件系统的所有者权限、用户组等信息进行查询和修改等。

在 php.ini 中激活“allow_url_open”选项后，可以使用 PHP 的 fopen() 或 fwrite() 等函数对远程文件进行操作。

- 读取或下载远程 Web 服务器的文件；
- 登录远程 FTP 服务器进行数据上传和下载；

7.2 Journal

用户使用 PHP 的 syslog() 函数还可以实现对分布式日志的管理和维护等。

7.3 File

除了可以针对 HTML 和文本文件进行处理之外，PHP 还可以对符合特定格式的文件进行快速处理。

- CVS
- INI
- XML

另外，PHP 也可以通过 zlib 来读取 ZIP 文件等。

在文件输出方面，PHP 支持输出 XHTML、XML、PDF、Flash 和图形文件等。

- PDF：PDFLib 库
- Flash：LibSWF 或 Ming 库
- 图像：GD 库

在 Windows 系统中，PHP 通过 Printer 扩展可以在服务器端开辟动态缓存空间来直接打印文件。

Chapter 8

Protocol

PHP 支持 FTP、LDAP、IMAP、SNMP、NNTP、POP3、HTTP 和 COM 等通信协议。

另外，PHP 还可以开放原始网络端口来使得任何其他的协议能够协同工作。

PHP 支持和所有 Web 开发语言之间的 WDDX 复杂数据交换，而且 PHP 支持对 Java 对象的使用，这样可以将它们自由地用作 PH 对象。

Chapter 9

Development

9.1 Interface

PHP 对 Apache 和 IIS 等服务器提供了直接的模块接口（SAPI），如果 PHP 不能作为模块支持 Web 服务器，还可以作为 CGI 或 FastCGI 处理器来使用，因此可以使用 PHP 的 CGI 可执行程序来处理所有服务器上的 PHP 文件请求。

- SAPI
- CGI

9.2 Kernel

PHP 允许开发者使用 C 语言开发新的功能模块，并且可以编译为 PHP 核心模块，这样就可以在运行时动态加载。

从本质上来说，PHP 的核心功能（例如字符串和数组处理）也是作为核心扩展来实现的，而且 PECL 提供了大量的第三方功能扩展。

9.3 PEAR

PEAR（PHP Extension and Application Repository）由 Stig S. Bakken 于 2000 年在 PHP 开发者会议上提出，目的是实做可以重复使用的库来提供 PHP 社区使用。

PEAR 库从 PHP4 开始投入开发，其目标为：

- 提供有架构的代码。
- 提供社区可重复使用的库。

- 创建 PHP 编码风格标准。

PEAR 库是 PHP 面向对象的应用和实践的最好例证，成熟的 PEAR 库包括：

表 9.1: PEAR 计划

Authentication	File Formats	Mail	Semantic Web
Benchmarking	File System	Math	Streams
Caching	Gtk Components	Networking	Structures
Configuration	Gtk2 Components	Numbers	System
Console	HTML	Payment	Text
Database	HTTP	PEAR	Tools and Utilities
Date & Time	Images	PHP	Validate
Encryption	Internationalization	Processing	Web Services
Event	Logging	Science	XML

9.4 PECL

除了内置的函数之外，PHP 也可以通过扩展库（extension）来提供数据库连接函数、数据压缩函数或图形处理等。

从 PHP5 开始引入了对全新的 PECL（PHP Extension Community Library）模块的支持，PECL 是 PEAR 打包系统的升级。

用户必须经过编译、安装和加载才能使用 PECL 扩展。例如，可以通过在 php.ini 中的 `extension` 指令加载扩展，或者用 `dl()` 函数。

9.4.1 pecl

PECL 网站中列出的 PECL 扩展库的发行版本可以用 pear 命令来下载和安装，也可以指明具体的版本号。

- `pecl download extname`

```
$ pecl download extname
```

- `pecl install extname`

```
$ pecl install extname
```

pecl 可以下载 extname 的源代码并进行编译，然后将 `extname.so` 安装到 extension_dir 中，这样就可以通过 php.ini 加载 `extname.so`[1]。

默认情况下，pecl 命令不会安装标记为 alpha 或 beta 状态的包。如果没有 stable 包可用，也可以用以下命令安装一个 beta 包：

```
$ pecl install extname-beta
```

也可以用此命令安装一个指定的版本：

```
$ pecl install extname-0.1
```

9.4.2 phpize

在手工编译 PECL 扩展时，phpize 命令用于准备 PHP 扩展库的编译环境。

在下面的例子中，扩展库的源程序位于 extname 目录中：

```
$ cd extname
$ phpize
$ ./configure
$ make
# make install
```

如果系统中没有 phpize 命令并且使用了预编译的包（例如 RPM），那要安装 PHP 包相应的开发版本，此版本通常包含了 phpize 命令以及相应的用于编译 PHP 及其扩展库的头文件，通过 `phpize --help` 命令可以显示此命令用法。

9.4.3 php-config

php-config 是一个简单的命令行脚本用于获取所安装的 PHP 配置的信息。在编译扩展时，如果安装有多个 PHP 版本，可以在配置时用 `--with-php-config` 选项来指定使用哪一个版本编译，该选项指定了相对应的 php-config 脚本的路径。

php-config 脚本在命令行所能使用的选项可以通过 `-h` 选项来显示：

```
Usage: /usr/local/bin/php-config [OPTION]
Options:
  --prefix         [...]
  --includes       [...]
  --ldflags        [...]
  --libs           [...]
  --extension-dir  [...]
```

[1]在 php.ini 中激活扩展之后，需要重新启动 web 服务以使更改生效。

```
--include-dir    [...]
--php-binary     [...]
--php-sapis      [...]
--configure-options [...]
--version        [...]
--vernum         [...]
```

表 9.2: php-config 脚本命令行选项

选项	说明
`--prefix`	PHP 所安装的路径前缀，例如 `/usr/local`
`--includes`	列出用 `-I` 选项包含的所有文件
`--ldflags`	PHP 编译时所使用的 `LD` 标志
`--libs`	PHP 编译时所附加的库
`--extension-dir`	扩展库的默认路径
`--include-dir`	头文件的默认路径前缀
`--php-binary`	PHP CLI 或者 CGI 可执行文件的完整路径
`--php-sapis`	列出所有可用的 SAPI 模块
`--configure-options`	重现当前 PHP 在编译时的配置选项
`--version`	PHP 版本号
`--vernum`	PHP 版本号，以整数表示

用户的扩展设定应该与所使用的 PHP 可执行文件的设定都保持一致，有些 Web 服务器就会搞混，因为其并不一定使用和 PHP 可执行文件处于同一目录下的 php.ini 文件。

要搞清楚具体使用了哪一个 php.ini 文件，在 phpinfo() 的输出中查看：

```
Configuration File (php.ini) Path C:\WINDOWS
Loaded Configuration File C:\Program Files\PHP\5.2\php.ini
```

激活一个扩展后，保存 php.ini 文件并重启动 web 服务器，然后用 phpinfo() 再次查看确定，此时新的扩展应该有其自己的一节。

或者在命令行运行：

```
drive:\\path\to\php\executable\php.exe -i
```

如果某扩展并未在 phpinfo() 中显示，应该查看日志以确定问题出在哪里。

- 如果是在命令行使用 PHP（CLI），扩展加载出错信息会直接在屏幕显示。
- 如果在 Web 服务器中使用 PHP，则日志文件的位置与格式各不相同。

最常见的问题是 DLL 文件的位置，php.ini 中“extension_dir”设定的值，以及编译时的设置不匹配。如果问题出在编译时设置不匹配，那可能所下载的 DLL 文件不对。可以尝试重新下载一个设置匹配的扩展。

阅读所使用的 Web 服务器之文档以确定日志文件的位置，这与 PHP 本身并无关系。此外，phpinfo() 可以起到很大帮助。

有时可能需要将扩展库静态编译到 PHP 中，此时就需要将扩展库源程序放入 `php-src/ext/` 目录中去并告诉 PHP 编译系统来生成其配置脚本。

```
$ cd /your/phpsrcdir/ext
$ pecl download extname
$ gzip -d < extname.tgz | tar -xvf -
$ mv extname-x.x.x extname
```

这将产生以下目录：

```
/your/phpsrcdir/ext/extname
```

此时强制 PHP 重新生成配置脚本[2]，然后正常编译 PHP：

```
$ cd /your/phpsrcdir
$ rm configure
$ ./buildconf --force
$ ./configure --help
$ ./configure --with-extname --enable-someotherext --with-foobar
$ make
$ make install
```

是否用 `--enable-extname` 或 `--with-extname` 取决于扩展库，不需要外部库文件的扩展库使用`--enable`。如果要确认的话，在 `buildconf` 之后运行：

```
$ ./configure -\/-help | grep extname
```

9.5 CGI

多线程 Web 服务器接收的所有请求都运行在相同的进程空间（Web 服务器自己的空间）中，该空间仅有一套环境变量。如果想获得 CGI 变量，例如 PATH_INFO、HTTP_HOST 等，使用原有的 PHP 3.x 的方式（getenv()），或使用类似的方式（注册全局变量到 $_ENV 环境变量）都是不可行的，只能获得运行中的 Web 服务器的环境变量，而不能获得任何有效的 CGI 变量。

[2]要运行“buildconf”脚本，需要 autoconf 2.13 和 automake 1.4+（更新版本的 autoconf 也许能工作，但不被支持）。

Web 服务器的启动脚本事实上是一个 CGI 脚本，因此启动脚本在启动 Web 服务器进程的同时就会包含一些 CGI 变量，但是如果从命令行手动启动 Web 服务器就不会产生这些 CGI 形式的变量。

PHP 默认情况下被编译为 CLI 和 CGI 程序，可以建立一个命令行解释器来处理 CGI 以及非 Web 相关的操作。如果需要为 Web 服务器提供 PHP 支持，那么通常为性能考虑应该使用模块方式。

CGI 可以使 Apache 用户用不同的用户 ID 运行不同的 PHP 页面，不过使用 CGI 方式进行服务器部署可能存在缺陷。

某些服务器提供的环境变量可能没有定义在当前的 CGI/1.1 标准中，只有下列变量定义在其中，其它的变量均作为“供应商扩展（vendor extensions）”来对待。

- AUTH_TYPE
- CONTENT_LENGTH
- CONTENT_TYPE
- GATEWAY_INTERFACE
- PATH_INFO
- PATH_TRANSLATED
- QUERY_STRING
- REMOTE_ADDR
- REMOTE_HOST
- REMOTE_IDENT
- REMOTE_USER
- REQUEST_METHOD
- SCRIPT_NAME
- SERVER_NAME
- SERVER_PORT
- SERVER_PROTOCOL
- SERVER_SOFTWARE

Chapter 10

Installation

在配置 PHP 应用服务器时，一般有两个方法可以将 PHP 连接到服务器上。

- SAPI
- CGI

PHP 向 Apache 和 IIS 等 Web 服务器提供了一个直接的模块接口 SAPI，其中 IIS 服务器支持微软的 ISAPI，可以极大地提高 IIS 的性能。

ISAPI 是驻留在 Web 服务器中的代码，而且和 Apache 模块的原理类似，ISAPI 在首次调用时就会被载入内存，这样在以后的使用中就不会启动新的进程。

如果 PHP 无法作为 Web 服务器模块，总是可以将其作为 CGI 或 FastCGI 处理器来使用，这样就可以使用 PHP 的 CGI 可执行程序来处理所有服务器上的 PHP 文件请求。

PHP 命令行脚本可以执行能够在离线状态下根据传递给脚本的参数，自动生成一些图片，或处理一些文本文件等工作。

10.1 UNIX

在 Unix 平台下安装 PHP 时可以使用配置和编译过程，或是使用各种预编译的包。

如果用户需要与标准配置不同的功能（例如安全服务器，或者不同的数据库驱动扩展模块），可能需要编译 PHP 和/或 Web 服务器。

为了直接从 Git 源文件或者自己修改过的包进行编译，可能需要某些辅助工具。

- autoconf：2.13+（PHP < 5.4.0），2.59+（PHP >= 5.4.0）
- automake：1.4+
- libtool：1.4.x+（除了 1.4.2）
- re2c：版本 0.13.4 或更高

- flex：版本 2.5.4（PHP <= 5.2）
- bison：版本 1.28（建议），1.35 或 1.75

Apache 初始的配置和安装过程由 configure 脚本中的一系列命令行选项控制，可以通过 `./configure --help` 命令了解 Apache 所有可用的编译选项及简短解释。

在配置好 Apache 后，下一步就可以使用 make 命令来编译模块和/或可执行文件。

```
# cd /src
# gzip -d httpd-2.4.12.tar.gz
# tar xvf httpd-2.4.12.tar
# cd httpd-2.4.12
# ./configure --help
```

一般情况下，可以直接使用 Aapche 的默认配置来进行编译。例如，这里将允许 Apache 加载 DSO（Dynamic Shared Object）模块：

```
# ./configure --enable-dso
# make
# make install
# /usr/local/apache2/bin/apachectl start
```

为了继续安装 PHP 和配置 Apache，可以暂时停止 Apahce 服务。

```
# /usr/local/apache2/bin/apachectl stop
```

在编译安装 PHP 之前，首先需要释放 PHP 的官方压缩包，然后进行 PHP 的配置。

```
# cd /src
# gzip -d php-5.6.10.tar.gz
# tar xvf php-5.6.10.tar
```

Apache 和 PHP 都被设计为模块化结构，因此可以配合各种不同的功能模块来进行工作，并且获得更多的扩展功能。

在编译 PHP 时可以手动配置编译选项来控制对应模块的使用，而且编译选项指定的模块必须事先已经安装，否则配置脚本将会报错。例如，下面的示例中中说明了编译 PHP 时的一些常见选项：

- `--prefix` 指定 PHP 的安装目录；
- `--with-apxs2` 用于提供对 Apache2 的动态共享模块的支持。

```
# cd php-5.6.10
# ./configure --prefix=/usr/local/php \
--with-apxs2=/usr/local/apache2/bin/apxs \
--with-zlib-dir=/usr/local/zlib2 \
--with-mysql=/usr/local/mysql \
--with-pgsql=/usr/local/pgsql \
```

在配置脚本检测通过后，就可以进行编译和安装，同样也可以重复上述步骤来改变编译选项。

```
# make
# make install
```

在编译并安装 PHP 之后，接下来将建立配置 php.ini，从而可以进一步修改 PHP 的选项。

默认情况下，配置阶段在/usr/local/lib 目录下建立 php.ini，因此可以使用 configure 命令的`--with-config-file-path=/path` 选项来手动指定。

```
# cp php.ini-production /usr/local/lib/php.ini
```

在 Apache 的 httpd.conf 中可以配置 PHP，从而确保 Apache 可以调用 PHP 模块和解析 PHP 脚本文件。

```
# cat >> httpd.conf
## 载入PHP5模块
LoadModule php5_module modules/libphp5.so
## 载入PHP4模块
LoadModule php4_module modules/libphp4.so

## 添加.php扩展名并进行解析
AddType application/x-httpd-php .php

## 添加.phps扩展来显示高亮的PHP源文件
AddType application/x-httpd-php-source .phps
```

在 Windows 操作系统中安装 MySQL 等数据库时，为了保持目录结构一致，同样也可以安装到 c:\usr\local\mysql，并且使用相应的配置工具来管理数据库服务。

在 MySQL 配置工具中，可以了解当前系统的运行状况，并提供系统环境、启动检测、服务器参数、配置文件、错误文件、系统变量、服务器进程、数据库和表结构以及综合报告等信息。

用户可以通过 MySQL 的配置文件 my.ini 对 MySQL 数据库服务器进行手动配置，并且在配置完毕后重启 MySQL 服务使其生效。

```
#This file was made using the WinMySQLAdmin Tool

[mysqld]
port=3306
basedir=c:/usr/local/mysql
#bind-address=192.168.1.10
```

```
datadir=c:/usr/local/mysql/data
#tmpdir=
#set-variable=key_buffer=16M

[WinMySQLAdmin]
Server=c:/usr/local/mysql/bin/mysql-nt.exe
user=root
password=p@ss
```

在 UNIX/Linux 操作系统中安装 MySQL 时，可以考虑使用 RPM 自动安装包，或者从二进制文件进行安装，不需要自己手动编译，只需要将安全文件包释放到 MySQL 目录中。

为了提高 MySQL 的灵活性，在安装之前可以为其建立对应的用户和用户组。

```
# addgroup mysql
# adduser -g mysql mysql
# cd src/
# gunzip < mysql-5.7.6-linux-glibc2.5-x86_64.tar.gz | tar xvf -
# ln -s mysql-5.7.6-linux-glibc2.5-x86_64 mysql
```

在 MySQL 的安装目录中包括程序文件和子目录等，而且必须将 MySQL 主目录添加到环境变量 PATH 中。

另外，在 MySQL 的子目录 scripts 中包含一个数据库初始化脚本，可以用来进行数据库初始化。

在使用 root 身份执行数据库初始化脚本 mysql_install_db 时，应该显式地使用“`--user`”指定 MySQL 的访问用户帐号。

```
# cd mysql
# ./scripts/mysql_install_db --user=mysql
```

除此之外，还需要让 root 用户和 mysql 用户对 MySQL 目录获得适当的权限。

```
# chown -R root /usr/local/mysql
# chown -R root /user/local/mysql/bin
# chown -R mysql /usr/local/mysql
# chgrp -R mysql /usr/local/mysql
```

在完成对 MySQL 的安装和设置后，可以启动 MySQL 数据库服务。

```
# /usr/local/mysql/bin/mysql_safe --user=mysql &
```

为了在操作系统启动时自动启动 MySQL 服务，可以将“support/mysql.server”文件复制到系统的启动目录中。

10.2 Mac OSX

Mac OS X 有内建支持的 PHP，编译安装类似于 Unix 系统下的安装，也可以使用预编译和打包的 PHP 版本来安装标准配置的 PHP。

如果需要不同的功能集（比如安全服务器或不同的数据库驱动程序），可能需要自己编译 PHP 和/或 Web 服务器。

10.2.1 Homebrew

Homebrew 是一个强大的 OS X 专用包管理器，它可以帮助你轻松的安装 PHP 和各种扩展。Homebrew PHP 是一个包含与 PHP 相关的 Formulae，能让你通过 homebrew 安装 PHP 的仓库。

也就是说, 你可以通过 brew install 命令安装 php53, php54, php55 或者 php56 ，并且通过修改 PATH 变量来切换各个版本。或者你也可以使用 brew-php-switcher 来自动切换。

10.2.2 MacPorts

MacPorts 是一个开源的，社区发起的项目，它的目的在于设计一个易于使用的系统，方便编译，安装以及升级 OS X 系统上的 command-line, X11 或者基于 Aqua 的开源软件。

MacPorts 支持预编译的二进制文件，因此你不必每次都重新从源码压缩包编译，如果你的系统没有安装这些包，它会节省你很多时间。

此时，你可以通过 port install 命名来安装 php53，php54，php55 或者 php56，比如：

```
$ sudo port install php54
$ sudo port install php55
```

也可以执行 select 命令来切换当前的 php 版本：

```
$ sudo port select --set php php55
```

10.2.3 PHPBrew

phpbrew 是一个安装与管理多个 PHP 版本的工具，在应用程序或者项目需要不同版本的 PHP 时非常有用，可以不再需要使用虚拟机来处理这些情况。

另外，php-osx.liip.ch 是另一种流行的选择，它提供了从 5.3 到 5.6 版本的单行安装功能。它并不会覆盖 Apple 集成的 PHP 文件，而是将其安装在了一个独立的目录中 (/usr/local/php5)。

最后一个可以用来控制安装 PHP 版本的选择就是自行编译。如果使用这种方法，必须先确认是否已经通过 Apple’s Mac Developer Center 下载、安装了 Xcode 或者“Command Line Tools for XCode”。

上述列出的解决方案主要是针对 PHP 本身，并不包含 Apache、Nginx 或者 SQL 服务器，使用 MAMP 或 XAMPP 等集成包可以安装这些软件并且将他们绑在一起，只是易于安装的背后也牺牲了一定的弹性。

在默认的 Web 服务器中启用 PHP 时，只需将 Apache 配置文件 `httpd.conf` 中的若干配置指令前面的注释符号去掉，这样 CGI 或 CLI 默认都可以使用（可以很容易的被终端程序使用）。

在搭建本地 PHP 开发环境时，标准的安装类型为 mod_php，接下来在 Apache Web 服务器中启用 PHP 时包含以下的步骤：

1. 找到并打开 Apache 的配置文件。

   默认情况下，这个配置文件的位置[1]是：`/private/etc/apache2/httpd.conf`。

2. 使用文本的编辑器取消注释。

```
# LoadModule php5_module libexec/httpd/libphp5.so
...
# AddModule mod_php5.c
```

3. 确保将所需要的文件扩展名解析为 PHP。

   确保将所需要的文件扩展名解析为 PHP（例如.php、.html 以及.inc），否则不能正常运行。

   自 Mac Panther 开始，将以下的配置写入 httpd.conf 后，只要 PHP 被启用就可以将.php 文件自动解析为 PHP 脚本。

```
<IfModule mod_php5.c>
    # If php is turned on, we respect .php and .phps files.
    AddType application/x-httpd-php .php
    AddType application/x-httpd-php-source .phps

    # Since most users will want index.php to work we
    # also automatically enable index.php
    <IfModule mod_dir.c>
        DirectoryIndex index.html index.php
    </IfModule>
</IfModule>
```

[1]在 Mac OS X 10.5 之前的版本中捆绑的是旧版本的 PHP 和 Apache。因此在旧的计算机中 Apache 配置文件的位置可能是 `/etc/httpd/httpd.conf`。

4. 确保 DirectoryIndex 加载了所需的默认索引文件。
   通常情况下，在 httpd.conf 中将 DirectoryIndex 设置为 index.php 和 index.html，可以根据实际情况进行相应的调整。
5. 设置 php.ini 的位置或者使用默认的位置。
   Mac OS X 上通常默认的位置是 `/usr/local/php/php.ini`，如果没有使用 php.ini，PHP 将使用所有的默认值。
6. 定位或者设置 DocumentRoot。
   DocumentRoot 是网站所有文件的根目录，该目录中的文件由 Web 服务器提供服务，从而使得 PHP 文件将在输出到浏览器之前解析为 PHP 脚本。
   通常情况下，Web 服务器的默认的路径是 /Library/WebServer/Documents，可以根据需要在 httpd.conf 中设置为任何其他目录。
   另外，用户自己的缺省 DocumentRoot 是 /Users/yourusername/Sites。
7. 创建 `phpinfo()` 文件。
   phpinfo() 将会显示 PHP 的相关系统信息。例如，可以在 DocumentRoot 下创建一个 PHP 文件，其代码如下：
   ```
   <?php phpinfo(); ?>
   ```
8. 重启 Apache，然后从浏览器访问 phpinfo 文件。
   要重启 Apache，可以在 shell 中执行 `sudo apachectl graceful`，也可以停止/启动 OS X 系统首选项中的“Personal Web Server”选项。
   默认情况下，从浏览器访问本地文件的 URL 一般类似于 `http://localhost/info.php`，或者使用`http://localhost/~yourusername/info.php` 来访问用户自己 DocumentRoot 中的文件。

CLI（或者旧版本中的 CGI）一般文件名为 php ，其路径可能是 /usr/bin/php。

用户可以打开一个终端并执行 `php -v` 来检查当前运行的 PHP 的版本，在脚本中调用 phpinfo() 也会显示相关的信息。

10.3 Windows

用户可以下载 PHP 的 Windows 二进制包，如果为 PHP 所在的根目录（php.exe 所在的文件夹）设置好环境变量（例如 PATH）就可以从命令行中直接执行 PHP。不过，在安装 PHP 之前需要确保系统中已经安装好 HTTP（Web）服务器，并确认其正常工作，而且不同的 PHP 版本的目录结构可能是不同的。

如果需要将生产环境部署在 Windows 上，那么 IIS7 可以提供最稳定和最佳的性能，

表 10.1: PHP5 和 PHP4 的文件对应关系

PHP5	PHP4	备注
php5ts.dll	php4ts.dll	核心动态库
ext/php_*.dll	extensions/php_.*dll	PHP 的扩展库文件
*.dll	dlls/*.dll	其他扩展文件
php5apache.dll	sapi/php4apache.dll	Apache1.3.x 模块
php5apache2.dll	sapi/php4apache2.dll	Apache 2.0.x 模块
php5isapi.dll	sapi/php4isapi.dll	ISAPI 模块，支持 IIS
php.exe、php-win.exe	cli/php.exe	CLI 可执行文件
php-cgi.exe	php.exe	CGI 可执行文件

用户可以使用 IIS7 的图形化插件 phpmanager 来设置并管理 PHP。

IIS7 有内置的 FastCGI ，只需要将 PHP 配置为它的处理器就可以使用。

10.3.1 Compile

如果有开发环境（例如 Microsoft Visual Studio），那么也可以编译 PHP 的源程序来进行安装。

- PHP 5.5.22/5.6.6 使用 Microsoft VC11 编译，不支持 Windows XP；
- PHP 5.4.38 使用 Microsoft VC9 编译，支持 Windows XP。

为了和 Linux 下的 PHP 安装目录匹配，可以将 PHP 的安装目录指定为 c:\usr\local\php，同时也需要将 PHP 的安装目录加入系统的 Path 环境变量中。

```
Windows c:\usr\local\php
UNIX/Linux /usr/local/php
```

在 Windows 系统下安装好 PHP 之后，可能还需要加载各种扩展库以实现更多的功能。

另外，虽然互联网上还有一些多合一的安装程序，但是它们没有一个是被 PHP 官方认可的，因此从http://www.php.net/downloads.php下载官方 Windows 安装包是系统安全和优化的最好选择，然后就可以为 IIS、PWS 或 Xitami 安装 CGI 版本的 PHP，并配置好 Web 服务器。

10.3.2 Install

PHP 官方安装程序支持两种安装类型（标准和高级），并且安装向导在安装过程中会收集足够的信息来设置 php.ini 文件，然后配置好 Web 服务器来使用 PHP。

- 标准：使用合理的默认配置进行安装；
- 高级：在配置中询问相关问题。

一般情况下，在 PHP 安装完成后会提示重新启动系统，并重启服务器，或直接开始使用 PHP。

10.3.3 Manual

虽然 Windows 安装程序是让 PHP 工作的最容易的方法，但是它有很多限制。例如，它不支持自动安装 PHP 扩展，因此使用安装程序安装 PHP 不是最佳方式，而且 PHP 官方安装程序不包含任何外部的 PHP 扩展（PHP_*.dll）。如果需要，可以在 Windows ZIP 包和 PECL 中找到。

旧版本的 PHP 安装程序不能自动配置 Apache，因此需要手动进行配置，而且使用 Windows 安装程序来安装的 PHP 不是安全的。

如果需要一个安全的 PHP 设置，最好使用手动方式安装，并手动设置好每个选项，这样就可以将 PHP 用于在线的生产服务器中。

PHP 的 Windows 安装程序是用 Wix 工具包（http://wix.sourceforge.net/）基于 MSI 建立的，使用这种方式安装 PHP 时，首先会提示选择要配置的 Web 服务器，以及所需的任何配置细节，然后它将安装并配置 PHP 和所有内置以及 PECL 扩展库，并且配置相关的 Web 服务器（例如 IIS、Apache 以及 Xitami）。

在安装并激活相关特性和扩展时，每个项目的下拉菜单中需要选择“Will be installed on local hard drive”来激活安装该单个项目。

如果选择“Entire feature will be installed on local hard drive”，将会安装所有该项目的子项（例如对“PDO”选择了此选项，则会安装所有的 PDO 驱动）[2]。

在接下来设定 Windows 使用的 PHP 和 php.ini 文件时，可以配置特定的 Web 服务器使用 PHP，也可以通过手工配置来支持其他的 Web 服务器。

新版本 PHP 的安装程序支持无声模式，有助于系统管理员更容易发布 PHP。例如，可以使用下面的命令来使用无声模式：

```
msiexec.exe /i php-VERSION-win32-install.msi /q
```

[2]事实上，并不推荐安装所有的扩展库，因为其中许多库还需要 PHP 之外的支持才能正常工作。不过，在安装好之后，可以从控制面板的程序和功能里可以调出更改模式来激活或关闭某扩展和功能。

在不同的服务器环境中，可以根据实际需要将安装目录作为命令行参数传递给安装程序，例如下面的命令将 PHP 安装到 c:\php：

```
msiexec.exe /i php-VERSION-win32-install.msi /q INSTALLDIR=c:\php
```

另外，可以用同样的语法指定 Apache 配置目录（APACHEDIR）、Sambar 服务器目录（SAMBARDIR）以及 Xitami 服务器目录（XITAMIDIR），而且还可以指定安装哪些特性。例如，下面的命令说明要安装 mysqli 扩展和 CGI 可执行程序：

```
msiexec.exe /i php-VERSION-win32-install.msi /q ADDLOCAL=cgi,ext_php_mysqli
```

在 Windows 中，可安装的 PHP 特性列表如下：

- MainExecutable - php.exe 可执行文件（默认已包含）
- ScriptExecutable - php-win.exe 可执行文件
- ext_php_* - 扩展库（例如 MySQL 是 ext_php_mysql）
- apache13 - Apache 1.3 模块
- apache20 - Apache 2.0 模块
- apache22 - Apache 2.2 模块
- apacheCGI - Apache CGI 可执行文件
- iis4ISAPI - IIS ISAPI 模块
- iis4CGI - IIS CGI 可执行文件
- iis4FastCGI - IIS CGI 可执行文件
- NSAPI - Sun/iPlanet/Netscape 服务器模块
- netserve - NetServe Web 服务器之 CGI 可执行文件
- Xitami - Xitami CGI 可执行文件
- Sambar - Sambar 服务器 ISAPI 模块
- CGI - php-cgi.exe 可执行文件
- PEAR - PEAR 安装程序
- Manual - CHM 格式的 PHP 手册

10.3.4 Update

如果要升级 PHP，可以正常使用图形模式或者从命令行运行安装程序，可以看到安装程序会读取当前安装的选项，移除旧的安装，并用同样选项重新安装 PHP，因此建议用这种方式更新 PHP 而不是手工替换安装目录下的文件。

10.3.5 Extend

在 Windows 下安装好 PHP 和 Web 服务器之后，可能想要安装一些扩展库来获得更多功能。

PHP 扩展库的 DLL 文件都具有 php_ 前缀，可以通过修改 php.ini 来选择当 PHP 启动时加载哪些扩展库，也可以在脚本中通过使用 `dl()` 来动态加载。

很多扩展库都内置于 Windows 版的 PHP 之中，这意味着要加载这些扩展库，额外的 DLL 文件和 extension 配置指令都不需要。

Windows 下的 PHP 扩展库列表列出了需要或曾经需要额外 PHP DLL 文件的扩展库，而且 PHP 搜索扩展库的默认位置在不同的 PHP 版本中是不同的。

- PHP 4 中是 drive:\php4\extensions；
- PHP 5 中是 drive:\php5。

  在 php.ini 文件[3]可以根据用户自己的 PHP 设置来修改扩展目录。
- 修改 extension_dir 设置以指向用户放置扩展库的目录或者说放置 php_*.dll 文件的位置。

  ```
  extension_dir = drive:\php\extensions
  ```
- 启用指定的扩展库，需要去掉 php.ini 中该行 `extension=php_*.dll` 前的注释符号 (;)。

  ```
  // 将这一行
  ;extension=php_bz2.dll

  // 改成这样
  extension=php_bz2.dll
  ```
- 有些扩展库需要额外的 DLL 才能工作。

  对于 Oracle（php_oci8.dll）等所需要的 DLL 没有绑定在发行包里的情况，可以将绑定的 DLL 从 drive:\php\dlls 拷贝到主目录 drive:\php 中，并且需要将 drive:\php 放到系统路径 PATH 中去。
- 某些 DLL 没有绑定在 PHP 发行包中，例如 PECL 提供的 PHP 扩展库需要单独下载。

  下表说明了哪些扩展库需要额外的 DLL。

表 10.2: PHP 扩展库

扩展库	说明	注解
php_bz2.dll	bzip2 压缩函数	无

[3]如果运行服务器模块版的 PHP，在修改了 php.ini 之后别忘了重新启动 web 服务器以使其改动生效。

扩展库	说明	注解
php_calendar.dll	Calendar 日历转换函数	自 PHP 4.0.3 起内置
php_crack.dll	Crack 密码破解函数	无
php_ctype.dll	ctype 家族函数	自 PHP 4.3.0 起内置
php_curl.dll	CURL，客户端 URL 库函数	需要：libeay32.dll，ssleay32.dll（已附带）
php_dba.dll	DBA：数据库（dbm 风格）抽象层函数	无
php_dbase.dll	dBase 函数	无
php_dbx.dll	dbx 函数	
php_domxml.dll	PHP 4 DOM XML 函数	PHP <= 4.2.0 需要：libxml2.dll（已附带），PHP >= 4.3.0 需要：iconv.dll（已附带）
php_dotnet.dll	.NET 函数	PHP <= 4.1.1
php_exif.dll	EXIF 函数	需要 php_mbstring.dll。并且在 php.ini 中，php_exif.dll 必须在 php_mbstring.dll 之后加载
php_fbsql.dll	FrontBase 函数	PHP <= 4.2.0
php_fdf.dll	FDF：表单数据格式化函数	需要：fdftk.dll（已附带）
php_filepro.dll	filePro 函数	只读访问
php_ftp.dll	FTP 函数	自 PHP 4.0.3 起内置
php_gd.dll	GD 库图像函数	在 PHP 4.3.2 中移除。此外注意在 GD1 中不能用真彩色函数，用 php_gd2.dll 替代。
php_gd2.dll	GD 库图像函数	GD2
php_gettext.dll	Gettext 函数	PHP <= 4.2.0 需要 gnu_gettext.dll（已附带），PHP >= 4.2.3 需要 libintl-1.dll，iconv.dll（已附带）
php_hyperwave.dll	HyperWave 函数	无
php_iconv.dll	ICONV 字符集转换	需要：iconv-1.3.dll（已附带），PHP >=4.2.1 需要 iconv.dll
php_ifx.dll	Informix 函数	需要：Informix 库
php_iisfunc.dll	IIS 管理函数库	无
php_imap.dll	IMAP，POP3 和 NNTP 函数	无
php_ingres.dll	Ingres II 函数	需要：Ingres II 库
php_interbase.dll	InterBase 函数	需要：gds32.dll（已附带）
php_java.dll	Java 函数	PHP <= 4.0.6 需要：jvm.dll（已附带）
php_ldap.dll	LDAP 函数	PHP <= 4.2.0 需要 libsasl.dll（已附带），PHP >= 4.3.0 需要 libeay32.dll，ssleay32.dll（已附带）
php_mbstring.dll	Multi-Byte String 多字节字符串函数	无
php_mcrypt.dll	Mcrypt 加密函数	需要：libmcrypt.dll
php_mhash.dll	Mhash 函数	PHP >= 4.3.0 需要：libmhash.dll（已附带）
php_mime_magic.dll	Mimetype 函数	需要：magic.mime（已附带）
php_ming.dll	Ming 函数（Flash）	无
php_msql.dll	mSQL 函数	需要：msql.dll（已附带）
php_mssql.dll	MSSQL 函数	需要：ntwdblib.dll（已附带）
php_mysql.dll	MySQL 函数	PHP >= 5.0.0 需要 libmysql.dll（已附带）
php_mysqli.dll	MySQLi 函数	PHP >= 5.0.0 需要 libmysql.dll（PHP <= 5.0.2 中是 libmysqli.dll）（已附带）
php_oci8.dll	Oracle 8 函数	需要：Oracle 8.1+ 客户端库
php_openssl.dll	OpenSSL 函数	需要：libeay32.dll（已附带）
php_overload.dll	PHP 4 对象过载函数	自 PHP 4.3.0 起内置，自 PHP 5.0.0 起移除

扩展库	说明	注解
php_pdf.dll	PDF 函数	无
php_pgsql.dll	PostgreSQL 函数	无
php_printer.dll	Printer 打印机函数	无
php_shmop.dll	Shared Memory 共享内存函数	无
php_snmp.dll	SNMP 函数	仅用于 Windows NT!
php_soap.dll	SOAP 函数	PHP >= 5.0.0
php_sockets.dll	Socket 函数	无
php_sybase_ct.dll	Sybase 函数	需要：Sybase 客户端库
php_tidy.dll	Tidy 函数	PHP >= 5.0.0
php_tokenizer.dll	Tokenizer 函数	自 PHP 4.3.0 起内置
php_w32api.dll	W32api 函数	无
php_xmlrpc.dll	XML-RPC 函数	PHP >= 4.2.1 需要 iconv.dll（已附带）
php_xslt.dll	XSLT 函数	PHP <= 4.2.0 需要 sablot.dll，expat.dll（已附带）。PHP >= 4.2.1 需要 sablot.dll，expat.dll，iconv.dll（已附带）。
php_yaz.dll	YAZ 函数	需要：yaz.dll（已附带）
php_zip.dll	Zip 文件函数	只读访问
php_zlib.dll	ZLib 压缩函数	自 PHP 4.3.0 起内置

10.3.6 Config

Apache 支持以 CGI 或 Apache 模块的方式运行 PHP，它们是基于对 Apache 不同的配置方案来完成的。

对 Apache 服务器的配置主要是通过修改 httpd.conf 来实现的。例如，如果需要以 CGI 方式运行 PHP5，那么需要向 httpd.conf 中添加如下的代码：

```
# 设置PHP目录的别名
ScriptAlias /php/ "c:/usr/local/php/''

# 关联特定的扩展名，以解析PHP文件
AddType application/x-httpd-php .php

# 设置使用PHP5执行CGI脚本的程序
Action application/x-httpd-php "/php/php-cgi.exe"
```

对于 PHP4，则需要在 httpd.conf 中添加下面的代码：

```
ScripteAlias /php/ "c:/usr/local/php/"
AddType application/x-httpd-php .php

# 设置使用PHP4执行CGI脚本的顺序
Action application/x-httpd-ph "/php/php.exe"
```

如果以 Apache 模块的方式来运行 PHP5，则需要在 httpd.conf 中添加下面的代码：

```
# 载入PHP5模块
LoadModule php5_module "c:/usr/local/php/php5apache2.dll"
AddType application/x-httpd-php .php

# 配置php.ini的路径
PHPIniDir "c:/usr/local/php"
```

如果 Apache 模块的方式来运行 PHP4，则需要将 php4apache2.dll 从 sapi 目录中拷贝到 PHP 安装目录下，并且在 httpd.conf 中添加下面的代码：

```
# 载入PHP4模块
LoadModule php4_module "c:/usr/local/php/php4apache2.dll"
AddType application/x-httpd-php .php

# 配置php.ini的路径
PHPIniDir "c:/usr/local/php"
```

默认情况下，Apache 提供的 Web 虚拟目录为 Apache 安装目录下的 htdocs 目录。

为了测试 PHP 在 Apache 服务器中运行是否正常，可以在 Web 虚拟目录（即 htdocs）中创建下面的测试文件，并且在浏览器中通过http://localhost/phpinfo.php进行访问。

```
<?php
phpinfo();
?>
```

IIS 支持以 CGI 或 ISAPI 模块的方式集成 PHP。

- 如果以 CGI 方式集成 PHP，可以将 PHP 映射到 CGI 执行程序（例如 php-cgi.exe）。
- 如果以 ISAPI 方式集成 PHP，可以将 PHP 映射到 ISAPI 模块（例如 php5isapi.dll）。

另外，如果用户需要使用 PHP 的 HTTP 认证功能，则需要在 ISAPI 筛选器中设置 PHP 的筛选器（例如 php5isapi.dll）。

10.3.7 Regedit

在 Windows 服务器上进行设置时也可以通过注册表来对 PHP 进行配置。

为了保护注册表安全，可以使用注册表文件的方式来向注册表中添加新内容。例如，下面的注册表文件可以用来以 ISAPI 方式配置 IIS 服务器：

```
Windows Registry Editor Version 5.0
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\w3svc\parameters\Script Map]
".php"="c:\\usr\\local\\php\\php5isapi.dll"
```

如果以 CGI 方式配置 PHP，则需要使用下面的注册表文件：

```
Windows Registry Editor Version 5.0
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\w3svc\parameters\Script Map]
".php"="c:\\usr\\local\\php\\php-cgi.exe"
```

在对 IIS 进行设置后，需要重启 IIS 服务器来使修改生效。

10.4 Docker

Chapter 11

Accelerator

11.1 Overview

通常情况下，PHP 都是解释执行的，不过也可以通过特定的工具（例如加速器）来将 PHP 代码编译为中间字节码来获得更好的性能。

在 Web 应用系统中，加速器本身是一个加快访问速度的代理服务器，使用硬件或软件来实现。

Web 加速器可以安装在客户端、移动设备以及 ISP 服务器上，并且对数据进行压缩等操作后传输给用户。

PHP 加速器可以被设计为 PHP 扩展来增强性能，而且大多数 PHP 加速器都是将 PHP 程序和模块编译为可直接执行的 opcode/bytecode 并缓存，这样在需要调用 PHP 来处理客户端请求可以直接读取来加快处理速度。

11.1.1 eAccelerator

为了更进一步提高性能，可以将 opcode/bytecode 载入内存缓冲区，这样就可以从内存直接执行，从而减小在运行时从硬盘读取和从内存拷贝的限制。

例如，从 Turck MMCache 衍生的 eAccelerator 可以将 PHP 程序、PHP 核心和相关库函数预先编译后缓存在共享内存中，在需要时可以被重复使用，从而实现加速 PHP 程序的目的。

```
# tar -jxvf eaccelerator.tar.gz
# cd
eaccelerator
# phpize
# ./configure--enable-eaccelerator=shared --with-php-config=/usr/bin/php-config
# make
# make install
```

默认情况下，新编译安装的模块位于/usr/lib64/php/modules/中。

为了预先加载扩展 PHP 模块，可以在/etc/ld.so.conf.d/中创建 ls.so.conf 文件，并新增如下配置:

```
# vim /etc/ld.so.conf.d/ld.so.conf
/usr/lib64/php/modules
# ldconfig
```

eAccelerator 根据当前版本的 PHP 核心进行编译，因此在更新 PHP 后需要手动更新 eAccelerator，并且每次都要在/etc/php.ini 中添加 eAccelerator 的配置信息，或者在/etc/php.d/中创建 eAccelerator 的配置文件。

```
;;;;;;;;;;;;;;;;;;;;
; http://eaccelerator.net/
; date:
;;;;;;;;;;;;;;;;;;;;
extension="eaccelerator.so''
eaccelerator.shm_size="16''
eaccelerator.cache_dir="/tmp/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"
```

如果要检查 eAccelerator 模块的运行状态，可以使用 phpinfo() 函数或 php -i 进行查询。

对于存放 eAccelerator 的临时文件的目录要注意设置适当的权限，并且在重启服务器才能使用 eAccelerator。

```
# mkdir /tmp/eaccelerator
# chmod 777 /tmp/eaccelerator
# apachectl restart
```

除了可以把将 PHP 编译产生的 bytecode 暂存在共享内存内重复使用来提高执行效率，Alternative PHP Cache（APC）还可以对中间码进行优化来和所有版本的 PHP 协同使用。

```
# pecl install APC
downloading APC-3.1.13.tgz ...
```

```
Starting to download APC-3.1.13.tgz （171, 591 bytes）
....done: 171,591 bytes
55 source files, building
running: phpize
Configuring for:
PHP Api Version: 20121113
Zend Module Api No: 20121212
Zend Extension Api No: 220121212
Enable internal debugging in APC [no] : yes
Enable per request file info about files used from the APC cache [no] : yes
Enable spin locks （EXPERIMENTAL） [no] : yes
Enable memory protection （EXPERIMENTAL） [no] : yes
Enable pthread mutexes （default） [no] : yes
Enable pthread read/write locks （EXPERIMENTAL） [yes] : yes
...
```

由 Lighttpd 计划提供的 XCache 解决了其他 opcacher 存在的问题（比如可以支持新的 PHP 版本），并且在 Linux 下可以支撑高负载状况，另外还支持 ThreadSafe/Windows。

11.2 Encoder

默认情况下，PHP 源代码是可以从服务器上直接读取的，这样虽然可以提高 PHP 的弹性，但是也相对会造成安全危机和性能下降等问题。

PHP 编码器可以对 PHP 代码进行编码来保护 PHP 源代码不被读取，也可以提升运行的性能，因此使用 PHP 编码器（例如 Zend 提供的脚本优化器）将 PHP 程序编译成字节码（byte code），然后再通过服务器上安装对应的程序来运行 PHP 脚本。

除了通过编码器加速之外，PHP 还可以通过动态的高速缓存机制来提升速度。

11.3 Engine

模板引擎让 PHP 应用程序可以实现逻辑和使用界面上的分离，让程序开发更容易进行。例如，Smarty 是使用 PHP 实现的模板系统，可以简化 PHP 应用程序的开发过程，其核心模板引擎可以通过标签来创建 Web 组件。

不过模板引擎存在性能方面的争议，因为 PHP 本身就是一个模板引擎，使用模板引擎反而变成“重新发明了轮子”。

模板引擎最主要的好处就是让不懂 PHP 代码的人也可以参与使用界面的开发，因为模板引擎的语言远比 PHP 简单。例如，MVC 模式就是对模板引擎的最好应用，从而可以让 PHP 编程人员可以和 HTML 前端程序员分工合作。

11.4 Compiler

Facebook 在 2010 年发布的 HipHop 编译器可以把 PHP 源代码编译成 C++ 来提高速度，并且降低 CPU 的负载。

11.5 WSGI

WSGI（Python Web Server Gateway Interface）是为 Python 语言定义的 Web 服务器和 Web 应用程序或框架之间的一种简单而通用的接口，现在许多其它语言中也出现了类似接口。

一般而言，Web 应用框架的选择将限制可用的 Web 服务器的选择，反之亦然。

最初，Python 应用程序通常是为 CGI、FastCGI、mod_python，或者特定 Web 服务器的自定义的 API 接口而设计的。

基于现存的 CGI 标准而设计的 WSGI 可以作为 Web 服务器与 Web 应用程序或应用框架之间的一种低级别的接口来方便可移植 Web 应用开发。具体来说，WSGI 可以划分两个部分，分别为“服务器”（或“网关”）和“应用程序”（或“应用框架”）。

在处理一个 WSGI 请求时，服务器会为应用程序提供环境信息及一个回调函数（Callback Function）。当应用程序完成处理请求后，通过前述的回调函数将结果回传给服务器。

WSGI 中间件同时实现了 API 的两方，因此可以在 WSGI 服务和 WSGI 应用之间起调和作用，因此从 WSGI 服务器的角度来说，中间件扮演应用程序，而从应用程序的角度来说，中间件扮演服务器。

WSGI“中间件”组件可以执行以下功能:

- 重写环境变量后，根据目标 URL 将请求消息路由到不同的应用对象。
- 允许在一个进程中同时运行多个应用程序或应用框架。
- 负载均衡和远程处理，通过在网络上转发请求和响应消息。
- 进行内容后处理（例如应用 XSLT 样式表）。

下面是用 Python 语言写的一个符合 WSGI 的“Hello World”应用程序示例。

```
def app（environ, start_response）:
  start_response（'200 OK', [（'Content-Type', 'text/plain'）]）
  yield "Hello world!\n"
```

- 第一行定义了一个名为 app 的 callable 并接受两个参数（environ 和 start_response），其中 environ 是一个字典包含了 CGI 中的环境变量，start_response 也是一个 callable，并接受两个必须的参数——status（HTTP 状态）和 response_headers（响应消息的头）。
- 第二行调用了 start_response，状态指定为“200 OK”，消息头指定为内容类型是“text/plain”。
- 第三行将响应消息的消息体返回。

下面是一个调用一个程序并获取它的应答消息的例子。

```
def call_application (app, environ) :
  body = []
  status_headers = [None, None]
  def start_response (status, headers) :
    status_headers[:] = [status, headers]
    return body.append
  app_iter = app (environ, start_response)
  try:
    for item in app_iter:
      body.append (item)
  finally:
    if hasattr (app_iter, 'close') :
    app_iter.close ()
  return status_headers[0], status_headers[1], ''.join (body)
status, headers, body = call_application (app, {...environ...})
```

Chapter 12

Runtime

12.1 php.ini

PHP 可以基于内置的配置选项进行配置，也可以在运行时动态指定。

PHP 的配置文件（php.ini）在 PHP 启动时被读取，在 php.ini 中可以使用环境变量。

- 对于服务器模块版本的 PHP，仅在 Web 服务器启动时读取一次。
  例如，对于 Apache 服务器模块版本的 PHP，仅在 Web 服务器启动时读取一次[1]。
- 对于 CGI 和 CLI 版本，每次调用都会读取。

PHP 的配置文件 php.ini 包含 PHP 运行时所需要的相关参数，并且在重启 Web 服务器后才能使新的 php.ini 设置生效。

php.ini 使用 Windows 的 INI 文件的书写风格，使用“;”作为注释。

另外，如果某些选项允许有多个值，那么应该使用系统列表分隔符（在 Windows 下是分号“;”，在 UNIX 下是冒号“:”）。

php.ini 的搜索路径如下（按顺序）：

- SAPI 模块所指定的位置（Apache 2 中的 PHPIniDir 指令，CGI 和 CLI 中的 `-c` 命令行选项，NSAPI 中的 `php_ini` 参数，THTTPD 中的 `PHP_INI_PATH` 环境变量）。
- PHPRC 环境变量。
- 如果为不同版本的 PHP 指定了不同的 php.ini 文件位置，将以下面的顺序检查注册表目录：`[HKEY_LOCAL_MACHINE\SOFTWARE\PHP\x.y.z]`，`[HKEY_LOCAL_MACHINE\SOFTWARE\PHP\x.y]` 和 `[HKEY_LOCAL_MACHINE\SOFTWARE\PHP\x]`，其中的 x，y 和 z 指的是 PHP 主版本号，次版本号和发行批次。如果在其中任何目录下的 IniFilePath 有键值，则第一个值将被用作 php.ini 的位置（仅适用于 Windows）。
- `[HKEY_LOCAL_MACHINE\SOFTWARE\PHP]` 内 IniFilePath 的值（Windows 注册表位置）。

[1] Apache 服务器在启动时会把目录转到根目录，这将导致 PHP 尝试在根目录下读取 php.ini。

- 当前工作目录（对于 CLI）。
- Web 服务器目录（对于 SAPI 模块）或 PHP 所在目录（Windows 下其它情况）。
- Windows 目录（`C:\Windows` 或 `C:\Winnt`），或 `--with-config-file-path` 编译时选项指定的位置。

12.2 php-SAPI.ini

如果存在 php-SAPI.ini（SAPI 是当前所用的 SAPI 名称，因此实际文件名为 php-cli.ini 或 php-apache.ini 等），则会用它替代 php.ini，SAPI 的名称可以用 `php_sapi_name()` 来测定。

由扩展库处理的 php.ini 指令的文档由扩展库提供，但是并非所有的 PHP 指令都有文档说明。

```
; any text on a line after an unquoted semicolon (;) is ignored
[php] ; section markers (text within square brackets) are also ignored
; Boolean values can be set to either:
;   true, on, yes
; or false, off, no, none
register_globals = off
track_errors = yes

; you can enclose strings in double-quotes
include_path = ".:/usr/local/lib/php"

; backslashes are treated the same as any other character
include_path = ".;c:\php\lib"
```

12.3 .user.ini

- 自 PHP 5.1.0 起，有可能在.ini 文件内引用已存在的.ini 变量，例如 `open_basedir = ${open_basedir} ":/new/dir"`。
- 自 PHP 5.3.0 起，PHP 支持基于每个目录的.htaccess 风格的 INI 文件，不过仅被 CGI/FastCGI SAPI 支持，并且导致 PECL 的 htscanner 扩展作废。如果使用 Apache，使用.htaccess 文件有同样效果。

除了主 php.ini 之外，PHP 还会在每个目录下扫描 INI 文件，从被执行的 PHP 文件所在目录开始一直上升到 Web 根目录（`$_SERVER['DOCUMENT_ROOT']` 所指定的）。如果被执行的 PHP 文件在 Web 根目录之外，则只扫描该目录。

在.user.ini 风格的 INI 文件中，只有具有 `PHP_INI_PERDIR` 和 `PHP_INI_USER` 模式的 INI 设置可被识别，其中 `user_ini.filename` 和 `user_ini.cache_ttl` 控制着用户 INI 文件的

使用。

- `user_ini.filename` 设定了 PHP 会在每个目录下搜寻的文件名，如果设定为空字符串则 PHP 不会搜寻，默认值是.user.ini。
- `user_ini.cache_ttl` 控制着重新读取用户 INI 文件的间隔时间，默认是 300 秒（5 分钟）。

这些模式决定着一个 PHP 的指令在何时何地，是否能够被设定，而且每个指令都有其所属的模式。

- 有些指令可以在 PHP 脚本中用 `ini_set()` 来设定;
- 有些指令只能在 php.ini 或 httpd.conf 中进行设定。

例如，`output_buffering` 指令是属于 `PHP_INI_PERDIR`，因而就不能用 `ini_set()` 来设定，但是 `display_errors` 指令是属于 `PHP_INI_ALL`，因而可以在任何地方被设定，包括 `ini_set()`。

表 12.1: PHP_INI_* 模式的定义

模式	含义
`PHP_INI_USER`	可在用户脚本（例如 `ini_set()`）或 Windows 注册表（自 PHP 5.3 起）以及.user.ini 中设定
`PHP_INI_PERDIR`	可在 php.ini，.htaccess 或 httpd.conf 中设定
`PHP_INI_SYSTEM`	可在 php.ini 或 httpd.conf 中设定
`PHP_INI_ALL`	可在任何地方设定

- PHP 运行于 Apache 模块方式
  当使用 PHP 作为 Apache 模块时，也可以用 Apache 的配置文件(例如 httpd.conf)和.htaccess 文件中的指令来修改 PHP 的配置设定，需要有“`AllowOverride Options`”或“`AllowOverride All`”权限才可以。
  有几个 Apache 指令可以使用户在 Apache 配置文件内部修改 PHP 的配置，至于哪些指令属于 `PHP_INI_ALL`，`PHP_INI_PERDIR` 或 `PHP_INI_SYSTEM` 中的哪一个，需要参考 php.ini 配置选项列表。
  - `php_value name value`
    设定指定的值。只能用于 `PHP_INI_ALL` 或 `PHP_INI_PERDIR` 类型的指令。要清除先前设定的值，把 value 设为 none。不要用 `php_value` 设定布尔值，应该用 `php_flag`。
  - `php_flag name on|off`
    用来设定布尔值的配置指令。仅能用于 `PHP_INI_ALL` 和 `PHP_INI_PERDIR` 类型的指令。
  - `php_admin_value name value`

设定指定的指令的值。不能用于.htaccess 文件。任何用 php_admin_value 设定的指令都不能被.htaccess 或 virtualhost 中的指令覆盖。要清除先前设定的值，把 value 设为 none。

- php_admin_flag name on|off
  用来设定布尔值的配置指令。不能用于.htaccess 文件。任何用 php_admin_flag 设定的指令都不能被.htaccess 或 virtualhost 中的指令覆盖。

```
<IfModule mod_php5.c>
 php_value include_path ".:/usr/local/lib/php"
 php_admin_flag engine on
</IfModule>
<IfModule mod_php4.c>
 php_value include_path ".:/usr/local/lib/php"
 php_admin_flag engine on
</IfModule>
```

PHP 常量不存在于 PHP 之外。例如在 httpd.conf 中不能使用 PHP 常量如 E_ALL 或 E_NOTICE 来设定 error_reporting 指令，因为其无意义，实际等于 0。应该用相应的掩码值来替代。这些常量可以在 php.ini 中使用。

- 通过 Windows 注册表修改 PHP 配置
  在 Windows 下运行 PHP 时，可以用 Windows 注册表以目录为单位来修改配置。
  PHP 配置值存放于注册表项 HKLM\SOFTWARE\PHP\Per Directory Values 下面，子项对应于路径名。例如，对于目录 c:\inetpub\wwwroot 的配置值会存放于 HKLM\SOFTWARE\PHP\Per Directory Values\c\inetpub\wwwroot 项下面。其中的设定对于任何位于此目录及其任何子目录的脚本都有效。项中的值的名称是 PHP 配置指令的名字，值的数据是字符串格式的指令值。值中的 PHP 常量不被解析。不过只有可修改范围是 PHP_INI_USER 的配置值可以用此方法设定，PHP_INI_PERDIR 的值就不行。
- 其它接口下的 PHP
  无论怎样运行 PHP，都可以在脚本中通过 ini_set() 而在运行时修改某个值。如果对自己系统中的配置设定及其当前值的完整列表感兴趣，可以运行 phpinfo() 函数并查看其结果的页面。也可以在运行时用 ini_get() 或 get_cfg_var() 取得个别配置指令的值。

12.4 directive

当 Web 服务器使用 CGI 方式运行 PHP 时，可以开启 cgi.force_redirect 来增强系统安全性，但是在 IIS 和 OmniHTTPD 等服务器时必须关闭。

PHP 超全局变量的解析顺序可以通过 variabls_order 进行设定，默认设定为 EGPCS，依

次为 $_ENV、$_GET、$_POST、$_COOKIE 和 $_SERVER。

基于安全的考虑，默认将 register_globals 设置为 Off 来禁止将 $_ENV、$_GET、$_POST、$_COOKIE 和 $_SERVER 变量同时设置为全局变量 $GLOBALS 的引用。

为了查找文件，可以使用 include_path 来指定一组目录以用于 require 或 include，以及 fopen_with_path() 函数。

```
## UNIX
include_path=".:/usr/local/php/pear"
## Windows
include_path=".;c:/usr/local/php/pear"
```

PHP 的扩展库可以使用 extension_dir 进行指定，通常是 PHP 安装目录下的 ext 目录。

为了指定 PHP 启动时加载的扩展库，可以使用 extension 进行指定。例如：

```
# UNIX/Linux
extension=curl.so
extension=ftp.so
# Windows
extension=php_curl.dll
extension=php_ftp.dll
```

Chapter 13

Security

攻击者无时无刻不在准备对 Web 应用程序进行攻击，因此提高 Web 应用程序的安全性是非常有必要的。

13.1 Remote files

只要在 php.ini 文件中激活了 allow_url_fopen 选项，就可以在大多数需要用文件名作为参数的函数中使用 HTTP 和 FTP 的 URL 来代替文件名，同时也可以在 include、include_once、require 及 require_once 语句中使用 URL。

例如，可以用以下示例的方式来打开远程 web 服务器上的文件，解析需要的输出数据，然后将这些数据用在数据库的检索中，或者简单地以和自己网站其它页面相同的风格输出其内容。

Example 18　获取远程文件的标题

```
<?php
$file = fopen ("http://www.example.com/", "r");
if (!$file) {
    echo "<p>Unable to open remote file.\n";
    exit;
}
while (!feof ($file)) {
    $line = fgets ($file, 1024);
    /* This only works if the title and its tags are on one line */
    if (eregi ("<title>(.*)</title>", $line, $out)) {
        $title = $out[1];
        break;
    }
```

```
}
fclose($file);
```

如果有合法的访问权限，以一个用户的身份和某 FTP 服务器建立了链接，还可以向该 FTP 服务器端的文件进行写操作，不过仅可以使用该方法来创建新的文件，如果尝试覆盖已经存在的文件，fopen() 函数的调用将会失败。

要以“anonymous”以外的用户名连接服务器，需要指明用户名以及密码（例如ftp://user:password@ftp.example.com/path/to/file），也可以在通过需要 Basic 认证的 HTTP 协议访问远程文件时使用相同的语法。

Example 19 上传数据到远程服务器

```
<?php
$file = fopen ("ftp://ftp.example.com/incoming/outputfile", "w");
if (!$file) {
    echo "<p>Unable to open remote file for writing.\n";
    exit;
}
/* Write the data here. */
fwrite ($file, $_SERVER['HTTP_USER_AGENT'] . "\n");
fclose ($file);
```

PHP 对远程文件的操作可以用来存储远程日志文件等，但是 fopen() 方式打开的 URL 中，仅能对新文件进行写操作。如果远程文件已经存在则 fopen() 函数的操作将会失败。

实际上，如果需要实现类似于分布式日志系统，需要使用 syslog() 函数等。

13.2 Authentication

PHP 的 HTTP 认证机制仅在 PHP 以 Apache 模块方式运行时才有效，因此该功能不适用于 CGI 版本。

在 Apache 模块的 PHP 脚本中，可以用 header() 函数来向客户端浏览器发送“Authentication Required”信息，使其弹出一个用户名/密码输入窗口。

当用户输入用户名和密码后，包含有 URL 的 PHP 脚本将会加上预定义变量 PHP_AUTH_USER，PHP_AUTH_PW 和 AUTH_TYPE 被再次调用，这三个变量分别被设定为用户名，密码和认证类型。预定义变量保存在 $_SERVER 数组中，支持“Basic”和“Digest”认证方法。

以下是在页面上强制客户端认证的脚本范例：

Example 20 Basic HTTP 认证

```
<?php
if(!isset($_SERVER['PHP_AUTH_USER'])){
  header('WWW-Authenticate: Basic realm="My Realm"');
  header('HTTP/1.0 401 Unauthorized');
  echo 'Text to send if user hits Cancel button';
  exit;
}else{
  echo '<p>Hello {$_SERVER['PHP_AUTH_USER']}.</p>';
  echo '<p>You entered {$_SERVER['PHP_AUTH_PW']} as your password';
}
```

Example 21 Digest HTTP 认证

```
<?php
$realm = 'Restricted area';

//user => password
$users = array('admin' => 'mypass', 'guest' => 'guest');


if (empty($_SERVER['PHP_AUTH_DIGEST'])) {
    header('HTTP/1.1 401 Unauthorized');
    header('WWW-Authenticate: Digest realm="'.$realm.
          '" qop="auth" nonce="'.uniqid().'" opaque="'.md5($realm).'"');

    die('Text to send if user hits Cancel button');
}


// analyze the PHP_AUTH_DIGEST variable
if (!($data = http_digest_parse($_SERVER['PHP_AUTH_DIGEST'])) ||
    !isset($users[$data['username']]))
    die('Wrong Credentials!');


// generate the valid response
$A1 = md5($data['username'] . ':' . $realm . ':' . $users[$data['username'
    ]]);
$A2 = md5($_SERVER['REQUEST_METHOD'].':'.$data['uri']);
$valid_response = md5($A1.':'.$data['nonce'].':'.$data['nc'].':'.$data['
    cnonce'].':'.$data['qop'].':'.$A2);
```

```
if ($data['response'] != $valid_response)
    die('Wrong Credentials!');

// ok, valid username & password
echo 'Your are logged in as: ' . $data['username'];


// function to parse the http auth header
function http_digest_parse($txt)
{
    // protect against missing data
    $needed_parts = array('nonce'=>1, 'nc'=>1, 'cnonce'=>1, 'qop'=>1, '
        username'=>1, 'uri'=>1, 'response'=>1);
    $data = array();

    preg_match_all('@(\w+)=([\'"]?)([a-zA-Z0-9=./\_-]+)\2@', $txt, $matches,
        PREG_SET_ORDER);

    foreach ($matches as $m) {
        $data[$m[1]] = $m[3];
        unset($needed_parts[$m[1]]);
    }

    return $needed_parts ? false : $data;
}
```

在编写 HTTP 标头代码时要格外小心。为了对所有的客户端保证兼容性，关键字“Basic”的第一个字母必须大写为“B”，分界字符串必须用双引号（不是单引号）引用，并且在标头行 `HTTP/1.0 401` 中，在 401 前必须有且仅有一个空格。

在实际运用中，可能需要对用户名和密码的合法性进行检查。或许进行数据库的查询，或许从 dbm 文件中检索。

有些 Internet Explorer 浏览器对标头的顺序要求严格，可以在发送 `HTTP/1.0 401` 之前首先发送 WWW-Authenticate 标头来可以解决该问题。

为了防止恶意用户通过编写脚本来从用传统外部机制认证的页面上获取密码，当外部认证对特定页面有效且安全模式被开启时，PHP_AUTH 变量将不会被设置，不过 REMOTE_USER 可以被用来辨认外部认证的用户，因此可以用 `$_SERVER['REMOTE_USER']` 变量来获取用户名。

PHP 内部通过是否有 AuthType 指令来判断外部认证机制是否有效，不过仍然不能防止有人通过未认证的 URL 来从同一服务器上认证的 URL 上偷取密码。

Netscape Navigator 和 Internet Explorer 浏览器都会在收到 401 的服务端返回信息时清空所有的本地浏览器整个域的 Windows 认证缓存，从而可以有效地注销一个用户，并强制用户重新输入用户名和密码，因此可以使用这种方法来使登录状态“过期”，或者作为“注销”按钮的响应行为。

Example 22　强制重新输入用户名和密码的 HTTP 认证

```
<?php
function authenticate() {
    header('WWW-Authenticate: Basic realm="Test Authentication System"');
    header('HTTP/1.0 401 Unauthorized');
    echo "You must enter a valid login ID and password to access this
        resource\n";
    exit;
}

if (!isset($_SERVER['PHP_AUTH_USER']) ||
    ($_POST['SeenBefore'] == 1 && $_POST['OldAuth'] == $_SERVER['
        PHP_AUTH_USER'])
) {
    authenticate();
} else {
    echo "<p>Welcome: {$_SERVER['PHP_AUTH_USER']}<br />";
    echo "Old: {$_REQUEST['OldAuth']}";
    echo "<form action='{$_SERVER['PHP_SELF']}' METHOD='post'>\n";
    echo "<input type='hidden' name='SeenBefore' value='1' />\n";
    echo "<input type='hidden' name='OldAuth' value='{$_SERVER['PHP_AUTH_USER
        ']}' />\n";
    echo "<input type='submit' value='Re Authenticate' />\n";
    echo "</form></p>\n";
}
```

该行为对于 HTTP 的 Basic 认证标准来说并不是必须的，因此不能依靠这种方法。例如，对 Lynx 浏览器的测试表明 Lynx 在收到 401 的服务端返回信息时不会清空认证文件，因此只要对认证文件的检查要求没有变化，只要用户点击“后退”按钮，再点击“前进”按钮，其原有资源仍然能够被访问。不过，用户可以通过按“_”键来清空他们的认证信息。

HTTP 认证无法工作在 IIS 服务器的 CGI 模式下，需要编辑 IIS 的设置“目录安全”，点击“编辑”并且只选择“匿名访问”，其它所有的复选框都应该留空。

在 IIS 的 ISAPI 模式下使用 PHP 4 的时候，无法使用 PHP_AUTH_* 变量，只能使用 HTTP_AUTHORIZATION。

```
<?php
list($user, $pw) = explode(':', base64_decode(substr($_SERVER['HTTP_AUTHORIZATION'], 6))
    );。
```

如果要 HTTP 认证能够在 IIS 下工作，PHP 配置选项 cgi.rfc2616_headers 必须设置成 0 (默认值)。

如果安全模式被激活，脚本的 UID 会被加到 WWW-Authenticate 标头的 realm 部分。

```
Cache-Control:no-store, no-cache, must-revalidate
Connection:keep-alive
Content-Security-Policy:script-src 'self' 'nonce-pagejs' 'nonce-global' 'unsafe-inline'
    'unsafe-eval'
Content-Type:text/html; charset=UTF-8
Date:Fri, 09 Dec 2016 09:26:43 GMT
Expires:Thu, 19 Nov 1981 08:52:00 GMT
Pragma:no-cache
Server:nginx/1.11.1
Strict-Transport-Security:max-age=31536000; includeSubdomains; preload
Transfer-Encoding:chunked
WWW-Authenticate:Basic realm='%E8%AF%B7%E8%BE%93%E5%85%A5%E8%B4%A6%E6%88%B7%E5%AF%86%E7%
    A0%81'
```

13.3 Password Hash

每个人在建构 PHP 应用时终究都会加入用户登录的模块，用户的帐号及密码会被储存在数据库中，在登录时用来验证用户。

在存储密码前正确的哈希密码是非常重要的。哈希密码是单向不可逆的，该哈希值是一段固定长度的字符串且无法逆向推算出原始密码。这就代表你可以哈希另一串密码来比较两者是否是同一个密码，但又无需知道原始的密码。如果不将密码哈希，那么当未授权的第三者进入你的数据库时，所有用户的帐号资料将会一览无遗。有些用户可能（很不幸的）在别的网站也使用相同的密码，所以务必要重视数据安全的问题。

password_hash 函数现在使用的是目前 PHP 所支持的最强大的加密算法 BCrypt，而且该函数未来会支持更多的加密算法。password_compat 库的出现是为了提供对 PHP >= 5.3.7 版本的支持。

在下面例子中，我们哈希一个字符串，然后和新的哈希值对比。因为我们使用的两个字符串是不同的（’secret-password’ 与 ‘bad-password’），所以登录失败。

```
<?php
require 'password.php';
```

```
$passwordHash = password_hash('secret-password',PASSWORD_DEFAULT);

if($password_verify('bad-password',$passwordHash)) {
  // correct password
} else {
  // wrong password
}
```

13.4 Variable filter

永远不要信任外部输入，务必在使用外部输入前进行过滤和验证。filter_var() 和 filter_input() 函数可以过滤文本并对格式进行校验（例如 email 地址）。

外部输入可以是任何东西：$_GET 和 $_POST 等表单输入数据，$_SERVER 超全局变量中的某些值，还有通过 `fopen('php://input', 'r')` 得到的 HTTP 请求体。

外部输入的定义并不局限于用户通过表单提交的数据，其他诸如上传和下载的文档、session 值、cookie 数据，还有来自第三方 Web 服务的数据等，这些都是外服输入。

虽然外部输入可以被存储、组合并在以后继续使用，但是它依旧是外部输入。每次处理、输出、连接或在代码中包含时，都需要提醒自己检查数据是否已经安全地完成了过滤。

数据可以根据不同的目的进行不同的过滤。比如，当原始的外部输入被传入到了 HTML 页面的输出当中，它可以在站点上执行 HTML 和 JavaScript 脚本（即跨站脚本攻击（XSS））。

一种避免 XSS 攻击的方法是在输出到页面前对所有用户生成的数据进行清理，使用 strip_tags() 函数来去除 HTML 标签或者使用 htmlentities() 或是 htmlspecialchars() 函数来对特殊字符分别进行转义从而得到各自的 HTML 实体。

另一个例子是传入能够在命令行中执行的选项。这是非常危险的（同时也是一个不好的做法），不过可以使用自带的 escapeshellarg() 函数来过滤执行命令的参数。

最后的一个例子是接受外部输入来从文件系统中加载文件。这可以通过将文件名修改为文件路径来进行利用。你需要过滤掉”/”, ”../”, null 字符或者其他文件路径的字符来确保不会去加载隐藏、私有或者敏感的文件。

13.5 Variable clean

数据清理是指删除（或转义）外部输入中的非法和不安全的字符。例如，需要在将外部输入包含在 HTML 中或者插入到原始的 SQL 请求之前对它进行过滤。当使用 PDO 中的限制参数功能时，它可以用来自动完成过滤的工作。

有些时候，可能需要允许一些安全的 HTML 标签输入进来并被包含在输出的 HTML 页面中，但这实现起来并不容易。尽管有一些像 HTML Purifier 的白名单类库为了这个原因

而出现，实际上更多的人通过使用其他更加严格的格式限制方式（例如使用 Markdown 或 BBCode）来避免出现问题。

验证是来确保外部输入的是你所想要的内容。比如，你也许需要在处理注册申请时验证 email 地址、手机号码或者年龄等信息的有效性。

13.6 Validity Verify

有效性验证是来确保外部输入的是应用程序实际所想要的内容。比如，用户需要在处理注册申请时验证 email 地址、手机号码或者年龄等信息的有效性。

13.7 Configure file

在应用程序创建配置文件时，最好的选择时参照以下的做法：

- 推荐将配置信息存储在无法被直接读取和上传的位置上。
- 如果一定要存储配置文件在根目录下，那么使用.php 的扩展名来进行命名。这将可以确保即使脚本被直接访问到，它也不会被以明文的形式输出出来。
- 配置文件中的信息需要被针对性的保护起来，对其进行加密或者设置访问权限。

13.8 Global Variable

register_globals 选项已经被移除并不再使用，同时也在提醒如果正在升级旧的应用程序的话，需要注意这一点。

当 register_globals 选项被开启时，它会使许多类型的变量（包括 $_POST, $_GET 和 $_REQUEST）被注册为全局变量，这样就很容易使程序无法有效地判断数据的来源并导致安全问题。例如，`$_GET['foo']` 可以通过 $foo 被访问到，也就是可以对未声明的变量进行覆盖。如果使用的 PHP 版本低于 5.4.0，务必确保 register_globals 是被设置为 off。

13.9 Error Reporting

虽然错误日志对于发现程序中的错误是非常有帮助的，但是有些时候它也会将应用程序的结构暴露给外部。

为了有效的保护应用程序不受到由此而引发的问题，需要将在你的服务器上使用开发和生产（线上）两套不同的配置。

为了在开发环境中显示所有可能的错误，将 php.ini 进行如下配置：

```
display_errors = On
display_startup_errors = On
```

```
error_reporting = -1
log_errors = On
```

将 error_reporting 的值设为 -1 将会显示出所有的错误，也包括在未来的 PHP 版本中新增加的类型和参数，-1 和最初的 E_ALL 是相同的。

E_STRICT 类型的错误已经被包含在了 E_ALL 中，如果需要在 5.3 中显示所有的错误信息，就需要使用 -1 或者 `E_ALL | E_STRICT`。

下面的示例说明了如何在不同版本的 PHP 中开启全部错误：

- < PHP 5.3

  -1 或 E_ALL
- PHP 5.3

  -1 或 E_ALL | E_STRICT
- > PHP 5.3

  -1 或 E_ALL

为了在生产环境中隐藏错误显示，将 php.ini 进行如下配置：

```
display_errors = Off
display_startup_errors = Off
error_reporting = E_ALL
log_errors = On
```

当在生产环境中使用这个配置时，错误信息依旧会被照常存储在 web 服务器的错误日志中，唯一不同的是将不再显示给用户。

13.10 Upload Handling

PHP 支持使用 POST 和 PUT 方法来上传文本和二进制文件，并且提供了认证和文件操作函数来完全控制允许哪些用户上传以及文件上传后怎样处理。

PHP 能够接受任何来自符合 RFC-1867 标准的 HTTP 客户端（包括浏览器）上传的文件。

php.ini 的 file_uploads , upload_max_filesize , upload_tmp_dirpost_max_size 以及 max_input_time 设置选项。

Example 23　文件上传表单

```
<!-- The data encoding type, enctype, MUST be specified as below -->
<form enctype="multipart/form-data" action="__URL__" method="POST">
    <!-- MAX_FILE_SIZE must precede the file input field -->
    <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
    <!-- Name of input element determines name in $_FILES array -->
    Send this file: <input name="userfile" type="file" />
```

```
    <input type="submit" value="Send File" />
</form>
```

13.10.1 memory_limit

如果内存限制设置被激活，可能需要将 memory_limit 设置的更大，需要确认 memory_limit 的设置的足够大。

13.10.2 post_max_size

如果 post_max_size 设置的值太小，则较大的文件会无法被上传，因此需要保证 post_max_size 的值足够大。

文件上传表单的属性必须是 `enctype="multipart/form-data"`，否则文件无法上传。

PHP 的 $_FILES 全局变量包含了所有上传的文件信息，以上范例产生的 $_FILES 数组的内容如下所示：

- `$_FILES['userfile']['name']` 客户端机器文件的原名称。
- `$_FILES['userfile']['type']` 文件的 MIME 类型（如果浏览器提供该信息，例如“image/gif”），PHP 默认不检查 MIME 类型。
- `$_FILES['userfile']['size']` 已上传文件的大小（单位为字节）。
- `$_FILES['userfile']['tmp_name']` 文件被上传后在服务端储存的临时文件名。
- `$_FILES['userfile']['error']` 和该文件上传相关的错误代码。
    1. `UPLOAD_ERR_OK` 值为 0 表示没有错误发生，文件上传成功。
    2. `UPLOAD_ERR_INI_SIZE` 值为 1 表示上传的文件超过了 php.ini 中 upload_max_filesize 选项限制的值。
    3. `UPLOAD_ERR_FORM_SIZE` 值为 2 表示上传文件的大小超过了 HTML 表单中 MAX_FILE_SIZE 选项指定的值。
    4. `UPLOAD_ERR_PARTIAL` 值为 3 表示文件只有部分被上传。
    5. `UPLOAD_ERR_NO_FILE` 值为 4 表示没有文件被上传。
    6. `UPLOAD_ERR_NO_TMP_DIR` 值为 6 表示找不到临时文件夹。
    7. `UPLOAD_ERR_CANT_WRITE` 值为 7 表示文件写入失败。

13.10.3 upload_tmp_dir

上传的文件默认会被储存到服务端的默认临时目录中，除非 php.ini 中的 upload_tmp_dir 设置为其它的路径。

服务端的默认临时目录可以通过更改 PHP 运行环境的环境变量 TMPDIR 来重新设置，但是在 PHP 脚本内部通过运行 putenv() 函数来设置是不起作用的。该环境变量也可以用来确认其它的操作是否也是在上传的文件上进行的。

Example 24　完成文件上传

```
<?php
$uploaddir = '/var/www/uploads/';
$uploadfile = $uploaddir . basename($_FILES['userfile']['name']);

echo '<pre>';
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
    echo "File is valid, and was successfully uploaded.\n";
} else {
    echo "Possible file upload attack!\n";
}

echo 'Here is some more debugging info:';
print_r($_FILES);

print "</pre>";
```

文件路径的表示方法有很多种，PHP 无法确保使用生僻外语的文件名（尤其是包含空格的）能够被正确的处理。

为了决定接下来要对该文件进行哪些操作，接受上传文件的 PHP 脚本应该实现任何逻辑上必要的检查，否则不对正在操作的文件进行验证就意味着用户能够访问其它目录下的敏感信息。

- `$_FILES['userfile']['size']` 变量可以排除过大或过小的文件；
- `$_FILES['userfile']['type']` 变量可以排除文件类型和某种标准不相符合的文件；
- `$_FILES['userfile']['error']` 变量可以根据不同的错误代码来计划下一步如何处理。

不过只能把这些当作一系列检查中的第一步，因为有些值完全由客户端控制而在 PHP 端并不检查，PHP 脚本要么将上传的文件从临时目录中删除，要么将其移动到其它的地方（例如上传到云存储）。

如果上传的文件没有被移动到其它地方也没有被改名，则该文件将在表单请求结束时被删除。

如果表单中没有选择上传的文件，则 `$_FILES['userfile']['size']` 的值将为 0，`$_FILES['userfile']['tmp` 将为空。

13.10.4　max_input_time

max_input_time 以秒为单位设定了脚本接收输入（例如文件上传）的最大时间。对于较大或多个文件或者用户的网速较慢时，可能会超过默认的 60 秒。

13.10.5 upload_max_filesize

MAX_FILE_SIZE 隐藏字段（单位为字节）必须放在文件输入字段之前，其值为接收文件的最大尺寸。这是对浏览器的一个建议，而且 PHP 也会检查此项，不过用户可以在 HTTP 客户端端简单绕过此设置，因此不要指望用此特性来阻挡大文件。

虽然 PHP 设置中的上传文件最大值不会失效，但是最好还是在表单中加上 MAX_FILE_SIZE 项目，这样可以避免用户在等待上传大文件上传之后才发现文件过大上传失败的错误。

MAX_FILE_SIZE 值不能大于 ini 设置中 upload_max_filesize 选项设置的值（默认为 2M 字节）。

13.10.6 max_execution_time

max_execution_time 设置的值太小时，如果脚本运行的时间超过该设置就会自动退出，因此需要保证 max_execution_time 设置的足够大。

max_execution_time 仅仅只影响脚本本身运行的时间，任何其它花费在脚本运行之外的时间（例如 system() 函数对系统的调用、sleep() 函数的使用、数据库查询、文件上传等），在计算脚本运行的最大时间时都不包括在内。

httpd 可能会丢弃从客户端获得的 content-type mime 头信息中第一个空格后所有的内容，因此 httpd 可能不支持文件上传特性。

13.11 Batch Uploading

PHP 的 HTML 数组特性支持文件类型，因此用户可以批量上传文件，例如：

Example 25 批量文件上传

```
<form enctype="multipart/form-data" action="__URL__" method="post">
<p>Pictures:
<input type="file" name="pictures[]" />
<input type="file" name="pictures[]" />
<input type="file" name="pictures[]" />
<input type="submit" value="Send" />
</p>
</form>
```

服务器端处理批量文件上传的逻辑如下：

```
<?php
foreach ($_FILES["pictures"]["error"] as $key => $error) {
    if ($error == UPLOAD_ERR_OK) {
```

```
        $tmp_name = $_FILES["pictures"]["tmp_name"][$key];
        $name = $_FILES["pictures"]["name"][$key];
        move_uploaded_file($tmp_name, "data/$name");
    }
}
```

PHP 支持同时上传多个文件并将它们的信息自动以数组的形式组织，因此可以对 input 域使用不同的 name 来上传多个文件。

为了实现批量上传，需要在 HTML 表单中对文件上传域使用和多选框与复选框相同的数组式提交语法，注意不应该将普通的输入字段和文件上传的字段混用同一个表单变量（例如都用 foo[]）。

Example 26　多文件批量上传

```
<form action="file-upload.php" method="post" enctype="multipart/form-data">
 Send these files:<br />
 <input name="userfile[]" type="file" /><br />
 <input name="userfile[]" type="file" /><br />
 <input type="submit" value="Send files" />
</form>
```

在以上表单被提交后，数组 `$_FILES['userfile']`，`$_FILES['userfile']['name']` 和 `$_FILES['userfile']['size']` 将被初始化，所有这些提交的信息都将被储存到以数字为索引的数组中。

例如，假设名为 /home/me/review.html 和 /home/me/xwp.out 的文件被提交，那么 `$_FILES['userfile']['name']` 的值将是 review.html，`$_FILES['userfile']['name'][1]` 的值将是 xwp.out。类似的，`$_FILES['userfile']['size']` 将包含文件 review.html 的大小，依此类推。

此外，也同时设置了 `$_FILES['userfile']['name'][0]`，`$_FILES['userfile']['tmp_name'][0]`，`$_FILES['userfile']['size'][0]` 以及 `$_FILES['userfile']['type'][0]`。

13.12 Request Method

13.12.1 GET

13.12.2 POST

13.12.3 PUT

PHP4 必须使用标准的输入流来读取一个 HTTP PUT 的内容。

Example 27　PHP4 使用 PUT 保存文件

```php
<?php
/* PUT data comes in on the stdin stream */
$putdata = fopen("php://stdin", "r");

/* Open a file for writing */
$fp = fopen("myputfile.ext", "w");

/* Read the data 1 KB at a time
   and write to the file */
while ($data = fread($putdata, 1024))
  fwrite($fp, $data);

/* Close the streams */
fclose($fp);
fclose($putdata);
```

PUT 请求比文件上传要简单的多，一般的形式为：

```
PUT /path/filename.text HTTP/1.1
```

远程客户端会将其中的 /path/filename.text 存储到 web 目录树，让 Apache 或者 PHP 自动允许所有人覆盖 web 目录树下的任何文件显然是很不明智的，因此要处理类似 PUT 的请求时必须先告诉 web 服务器需要用特定的 PHP 脚本来处理该请求。

在 Apache 下，可以用 Script 选项来设置，通常在 <Directory> 区域或者 <Virtualhost> 区域进行设置。

```
Script PUT /put.php
```

上述设置指示 Apache 将所有对 URI 的 PUT 请求全部发送到 put.php 脚本，这些 URI 必须和 PUT 命令中的内容相匹配。

在 put.php 文件中，可以作如下操作：

```php
<?php
copy($PHP_UPLOADED_FILE_NAME, $DOCUMENT_ROOT . $REQUEST_URI);
```

把文件拷贝到远程客户端请求的位置时，可能希望在文件拷贝之前进行一些检查或者对用户认证之类的操作。

这里唯一的问题是，当 PHP 接受到 PUT 方法的请求时，它将会把上传的文件储存到和其它用 POST 方法处理过的文件相同的临时目录。在请求结束时，临时文件将被删除，因此用来处理 PUT 的 PHP 脚本必须将该文件拷贝到其它的地方。

PUT 上传的临时文件的文件名被储存在变量 $PHP_PUT_FILENAME 中，也可以通过 $REQUEST_URI 变量获得建议的目标文件名（在非 Apache web 服务器上可能会有较大的变

化)。该目标文件名是由远程客户端指定的，也可以不遵循客户端提供的信息，把所有上传的文件存储到一个特殊的上传目录下。

13.13 Connection handling

在 PHP 内部，系统维护着连接状态，有三种可能的状态:

- 0 - NORMAL（正常）
- 1 - ABORTED（异常退出）
- 2 - TIMEOUT（超时）

当 PHP 脚本正常地运行 NORMAL 状态时，连接为有效。当远程客户端中断连接时，ABORTED 状态的标记将会被打开。远程客户端连接的中断通常是由用户点击 STOP 按钮导致的。当连接时间超过 PHP 的时限时，TIMEOUT 状态的标记将被打开。

用户可以决定脚本是否需要在客户端中断连接时退出，不过有时候让脚本完整地运行会带来很多方便，即使没有远程浏览器接受脚本的输出。

默认的情况是当远程客户端连接中断时脚本将会退出，该处理过程可由 php.ini 的 ignore_user_abort 或由 httpd.conf 设置中对应的“php_value ignore_user_abort”以及 ignore_user_abort() 函数来控制。

如果没有告诉 PHP 忽略用户的中断，脚本将会被中断，除非通过 register_shutdown_function() 设置了关闭触发函数。

通过该关闭触发函数，当远程用户点击 STOP 按钮后，脚本再次尝试输出数据时，PHP 将会检测到连接已被中断，并调用关闭触发函数。

脚本也有可能被内置的脚本计时器中断，默认的超时限制为 30 秒。同样地，这个值可以通过设置 php.ini 的 max_execution_time 或 httpd.conf 设置中对应的“php_value max_execution_time”参数或者 set_time_limit() 函数来更改。当计数器超时的时候，脚本将会类似于以上连接中断的情况退出，先前被注册过的关闭触发函数也将在这时被执行。

在关闭触发函数中，可以通过调用 connection_status() 函数来检查超时是否导致关闭触发函数被调用。如果超时导致了关闭触发函数的调用，该函数将返回 2。

需要注意的一点是，ABORTED 和 TIMEOUT 状态可以同时有效，这样的设置在告诉 PHP 忽略用户的退出操作时是可能的，从而让 PHP 仍然注意用户已经中断了连接但脚本仍然在运行的情况。如果到了运行的时间限制，脚本将被退出，设置过的关闭触发函数也将被执行。在这时会发现函数 connection_status() 返回 3。

13.14 Persistent connections

持久的数据库连接是指在脚本结束运行时不关闭的连接。当收到一个持久连接的请求时。PHP 将检查是否已经存在一个（前面已经开启的）相同的持久连接。

- 如果存在，将直接使用这个连接；
- 如果不存在，则建立一个新的连接。

这里，所谓“相同”的连接是指用相同的用户名和密码到相同主机的连接。

没有完全理解 Web 服务器的工作原理和分布式负载的用户可能会错误地理解持久连接的作用，持久连接没有在相同的连接上提供建立“用户会话”的能力，也不提供有效建立事务的能力。

实际上，从严格意义上来讲，持久连接不会提供任何非持久连接无法提供的特殊功能，原因和 Web 服务器工作的方式有关。

Web 服务器可以用三种方法来利用 PHP 生成 web 页面。

第一种方法是将 PHP 用作一个单独运行的语言解释器 (CGI Wapper)。以这种方法运行，PHP 会为向 web 服务器提出的每个 PHP 页面请求生成并结束一个 PHP 解释器线程。由于该线程会随每个请求的结束而结束，因此任何在这个线程中利用的任何资源（例如指向 SQL 数据库服务器的连接）都会随线程的结束而关闭。在这种情况下，使用持久连接和非持久连接没有任何区别——因为 PHP 脚本本身的执行不是持久的。

第二种方法是把 PHP 用作多进程 web 服务器的一个模块（只适用于 Apache）。对于一个多进程的服务器，其典型特征是有一个父进程和一组子进程协调运行，其中实际生成 web 页面的是子进程。每当客户端向父进程提出请求时，该请求会被传递给还没有被其它的客户端请求占用的子进程。这也就是说当相同的客户端第二次向服务端提出请求时，它将有可能被一个不同的子进程来处理。在开启了一个持久连接后，所有请求 SQL 服务的后继页面都能够重用这个已经建立的 SQL Server 连接。

第三种方法是将 PHP 用作多线程 web 服务器的一个插件，PHP 对 ISAPI、WSAPI 和 NSAPI 的支持使得 PHP 可以被用作多线程 Web 服务器的插件，持久连接的行为和前面所描述的多过程模型在本质上是相同的。

持久连接所能提供的好处仅仅是效率，当 Web Server 创建到 SQL 服务器的连接耗费 (Overhead) 较高（如耗时较久，消耗临时内存较多）时，持久连接将更加高效。Overhead 高低取决于很多因素（例如数据库的种类，数据库服务和 web 服务是否在同一台服务器上以及 SQL 服务器负载状况等)。当 Overhead 较高，每次创建数据库连接成本较高时，持久连接将显著的提高效率。

持久连接使得每个子进程在其生命周期中只做一次连接操作，而非每次在处理一个页面时都要向 SQL 服务器提出连接请求。这也就是说，每个子进程将对服务器建立各自独立的持久连接。例如，如果有 20 个不同的子进程运行某脚本建立了持久的 SQL 服务器持久连接，那么实际上向该 SQL 服务器建立了 20 个不同的持久连接，每个进程占有一个。

如果持久连接的子进程数目超过了设定的数据库连接数限制，系统将会产生一些问题。如果数据库的同时连接数限制为 16，而在繁忙会话的情况下，有 17 个线程试图连接，那么有一个线程将无法连接。如果这个时候，在脚本中出现了使得连接无法关闭的错误（例如无限循环)，则该数据库的 16 个连接将迅速地受到影响，因此需要即时处理已放弃的以及闲置

的连接。

在使用持久连接时还有一些特别的问题需要注意。例如，

在持久连接中使用数据表锁时，如果脚本不管什么原因无法释放该数据表锁，其随后使用相同连接的脚本将会被持久的阻塞，这种情况下需要重新启动 httpd 服务或者数据库服务。

在持久连接中使用事务处理时，如果脚本在事务阻塞产生前结束，则该阻塞也会影响到使用相同连接的下一个脚本，这种情况下同样需要重新启动 httpd 服务或者数据库服务。

不管在什么情况下，都可以通过使用 register_shutdown_function() 函数来注册一个简单的清理函数来打开数据表锁，或者回滚事务，实际上更好的处理方法是不在使用数据表锁或者事务处理的脚本中使用持久连接，这样就可以从根本上解决这个问题（在其它地方仍然可以使用持久连接）。

持久连接与常规的非持久连接应该是可以互相替换的，即将持久连接替换为非持久连接时，脚本的行为也不应该发生改变，持久连接只应该改变脚本的效率而非其行为。

13.15 Thread Safe

Windows 版的 PHP 从版本 5.2.1 开始有 Thread Safe（线程安全）和 None Thread Safe（NTS，非线程安全）之分，这两者不同在于何处[?]？到底应该用哪种？

从 2000 年 10 月 20 日发布的第一个 Windows 版的 PHP3.0.17 开始的都是线程安全的版本。

与 Linux/Unix 系统是采用多进程的工作方式不同的是，Windows 系统的是多线程的工作方式，因此在 IIS 下以 CGI 方式运行 PHP 就会非常慢，原因就是 CGI 模式是建立在多进程的基础上的，而非多线程。

一般情况下，在 Windows 平台上需要把 PHP 配置成以 ISAPI 的方式来运行，ISAPI 是多线程的方式，可以加快运行速度。

不过，仍然存在一个问题，那就是很多常用的 PHP 扩展是以 Linux/Unix 的多进程思想来开发的，这些扩展在 ISAPI 的方式运行时就会出错并拖垮 IIS，因此在 IIS 下 CGI 模式才是 PHP 运行的最安全方式。

CGI 模式的不足就是对于每个 HTTP 请求都需要重新加载和卸载整个 PHP 环境，其资源消耗是巨大的。

为了兼顾 IIS 下 PHP 的效率和安全，微软给出了 FastCGI 的解决方案来让 PHP 的进程重复利用而不是每一个新的请求就新建一个进程，同时 FastCGI 也可以允许几个进程同时执行，从而既解决了 CGI 进程模式消耗太大的问题，又利用了 CGI 进程模式不存在线程安全问题的优势。

综合来说，如果使用 ISAPI 的方式来运行 PHP 就必须用 Thread Safe（线程安全）的版本，FastCGI 模式运行 PHP 没有必要进行线程安全检查，None Thread Safe（非线程安全）的

版本能够更好的提高效率。

- 在 Windows 版本的 Apache 下运行 PHP 时需要 VC6 版本的 PHP；
- 在 IIS 下运行 PHP 则需要 VC9 版本的 PHP。

13.16 Safe Mode

PHP 的安全模式是为了试图解决共享服务器（shared-server）安全问题而设立的，现在已经废弃（强制开启会报告 E_CODE_ERROR 级别的错误）。

虽然在结构上试图从 PHP 层上解决这个问题是不合理的，但是修改 web 服务器层和操作系统层显得非常不现实，因此 ISP 大量地使用安全模式。

表 13.1: 保安措施和安全模式配置指令

名字	默认	可修改范围	说明
safe_mode	"0"	PHP_INI_SYSTEM	已废弃
safe_mode_gid	"0"	PHP_INI_SYSTEM	已废弃
safe_mode_include_dir	NULL	PHP_INI_SYSTEM	已废弃
safe_mode_exec_dir	""	PHP_INI_SYSTEM	已废弃
safe_mode_allowed_env_vars	"PHP_"	PHP_INI_SYSTEM	已废弃
safe_mode_protected_env_vars	"LD_LIBRARY_PATH"	PHP_INI_SYSTEM	已废弃

PHP 默认不会运行在安全模式下，PHP 安全模式设置的限制不适用于可执行文件。

13.16.1 safe_mode

php.ini 中的 safe_mode 设置是否启用安全模式。

13.16.2 safe_mode_gid

php.ini 中的 safe_mode_gid 配置是否把 UID 比较检查放宽到 GID 比较检查。

默认情况下，安全模式在打开文件时会进行 UID 比较检查，打开 safe_mode_gid 可以放宽到 GID 比较。是否在文件访问时使用 UID（FALSE）或者 GID（TRUE）来做检查。

13.16.3 safe_mode_exec_dir

在安全模式下，可以通过 safe_mode_exec_dir 设置具有系统程序执行功能的目录，而且 exec()、system() 或 popen() 等函数只能调用执行该目录下的系统程序。

如果 PHP 使用了安全模式，system() 和其它程序执行函数将拒绝启动不在此目录中的程序，必须使用 / 作为目录分隔符（包括 Windows 中）。

13.16.4 safe_mode_include_dir

在安全模式下，safe_mode_include_dir 可以设置严格限制进行文件包含的目录，如果一个 PHP 程序使用 include 或 require 包含该目录（及子目录）中的另一个文件时，必须保证彼此具有相同的 UID 或 GID。

当从此目录及其子目录（目录必须在 include_path 中或者用完整路径来包含）包含文件时越过 UID/GID 检查，而且 safe_mode_include_dir 指令可以接受和 include_path 指令类似的风格用冒号（Windows 中是分号）隔开的路径，而不只是一个目录。

指定的限制实际上是一个前缀，而非一个目录名。也就是说，“safe_mode_include_dir = /dir/incl”将允许访问“/dir/include”和“/dir/incls”，如果它们存在的话。如果希望将访问控制在一个指定的目录，那么在结尾加上一个斜线（例如“safe_mode_include_dir = /dir/incl/”。如果本指令的值为空，具有不同 UID/GID 的文件将不能被包含。

13.16.5 safe_mode_allowed_env_vars

设置某些环境变量可能是潜在的安全缺口。本指令包含有一个逗号分隔的前缀列表。

在安全模式下，用户只能改变那些名字具有在这里提供的前缀的环境变量。默认情况下，用户只能设置以 PHP_ 开头的环境变量（例如 PHP_FOO = BAR）。

如果本指令为空，PHP 将使用户可以修改任何环境变量

13.16.6 safe_mode_protected_env_vars

safe_mode_protected_env_vars 指令包含有一个逗号分隔的环境变量的列表，最终用户不能用 putenv() 来改变这些环境变量，在 safe_mode_allowed_env_vars 中设置了允许修改时也不能改变这些变量。

如果 safe_mode 设置为 on，PHP 将通过文件函数或其目录检查当前脚本的拥有者是否和将被操作的文件的拥有者相匹配。

```
-rw-rw-r--  1 rasmus  rasmus     33 Jul 1 19:20 script.php
-rw-r--r--  1 root    root     1116 May 26 18:01 /etc/passwd
```

如果安全模式被激活，则执行 `readfile('/etc/passwd');` 将会导致以下错误：

```
Warning: SAFE MODE Restriction in effect. The script whose uid is 500 is not
allowed to access /etc/passwd owned by uid 0 in /docroot/script.php on line 2
```

在某些环境下，宽松的 GID 检查已经足够，严格的 UID 检查反而是不适合的，因此可以用 safe_mode_gid 选项来控制这种检查。如果设置为 On 则进行宽松的 GID 检查，设置为 Off（默认值）则进行 UID 检查。

13.16.7 disable_functions

PHP 安全模式允许单独屏蔽某些函数，但是 disable_functions 选项不能在 php.ini 文件外部使用，也就是说无法在 httpd.conf 文件的按不同虚拟主机或不同目录的方式来屏蔽函数。例如，如果将如下内容加入到 php.ini 文件：

```
disable_functions readfile,system
```

如果在设置了 disable_functions 选项后运行同样的 script.php，则其结果如下：

```
Warning: readfile() has been disabled for security reasons in
/docroot/script.php on line 2
```

13.16.8 open_basedir

除了 safe_mode 以外，如果设置了 open_basedir 选项，则所有的文件操作将被限制在指定的目录下。例如：

```
<Directory /docroot>
 php_admin_value open_basedir /docroot
</Directory>
```

如果在设置了 open_basedir 选项后运行同样的 script.php，则其结果如下：

```
Warning: open_basedir restriction in effect. File is in wrong directory in
/docroot/script.php on line 2
```

无论是否打开安全模式，open_basedir 都可以设置能够进行的文件操作的目录或文件的前缀，可以使用系统分隔符（“;”或“:”）添加多个列表。

当一个脚本试图使用 fopen() 或 gzopen() 打开一个文件时，那么该文家的位置将被检查，而且这个限制是前缀，不是目录名。

如果基于安全原因需要禁用某些函数/类时，可以使用 disable_functions/disable_classes，并且接受逗号分隔的函数/类名列表作为参数，不受安全模式的影响。

Part II

FastCGI

Chapter 14

Overview

如果 Web 服务器不支持以服务器模块（例如 Apache）的方式运行 PHP 等脚本语言，就必须考虑使用 CGI 或 FastCGI 的方式。

14.1 CGI

CGI（Common Gateway Interface，通用网关接口）是一种描述了服务器和请求处理程序之间传输数据的标准，可以让一个客户端从网页浏览器向执行在网络服务器上的程序请求数据。

CGI 最初由 NCSA（美国国家超级电脑应用中心）在 1993 年为 NCSA HTTPd Web 服务器开发，在 NCSA HTTPd Web 服务器中使用了 UNIX shell 环境变量来保存从 Web 服务器传递出去的参数，然后生成一个运行 CGI 的独立的进程。

CGI 的目标之一是要独立于任何语言，并且要求 Web 服务器无须在这个问题上对语言有任何了解，因此 CGI 程序可以用任何脚本语言或者是完全独立的编程语言实现，只要这个语言可以在这个系统上运行。例如，Unix shell script, Python, Ruby, PHP, Tcl, C/C++ 和 Visual Basic 等都可以用来开发 CGI 程序。

下面以实现维基百科编辑的 CGI 程序为例进行说明，首先用户代理程序（例如浏览器）向这个 CGI 程序请求某个名称的条目，如果该条目页面存在，CGI 程序就会去获取那个条目页面的原始数据，然后把它转换成 HTML 并把结果输出给浏览器；如果该条目页面不存在，CGI 程序则会提示用户新建一个页面。所有维基操作都是通过这个 CGI 程序来处理的。

从 Web 服务器的角度看，CGI 的工作方式就是在特定的位置（比如`http://www.example.com/wiki.cgi`）定义了可以运行 CGI 程序，这样当收到一个匹配 URL 的请求时，相应的程序就会被调用，并将客户端发送的数据作为输入。程序的输出会由 Web 服务器收集，并加上合适的 HTTP 头，再发送回客户端。

一般情况下，每次的 CGI 请求都需要新生成一个程序的副本来运行，这样大的工作量

会很快将服务器压垮，因此通常是使用更有效的技术（例如 mod_perl）来让脚本解释器直接作为模块集成在 Web 服务器（例如 Apache）中，从而避免重复载入和初始化解释器。

不过，这只是针对那些需要解释器的高级语言（即解释语言）而言的，使用 C 语言等编译型语言则可以避免这种额外负荷，而且 C 语言程序运行速度更快、对操作系统的负荷更小，因此使用编译型语言程序可以达到更高的执行效率。

如果代码的变动很少，那么可以在服务器产生一个新的进程在编译代码之前进行代码转译处理。例如，FastCGI 会在第一次调用脚本时，在系统的某个地方保存脚本编译过的版本，这样对这个文件以后的请求就会自动转向这个编译过的代码，而不用每次调用脚本解释器来解释脚本。当更改了代码时，需要清空临时缓存，就可以保证调用的是最新的版本的脚本代码。

除了存储中间转译结果之外，还可以直接把解释器集成在 Web 服务器中，这样就无须新建一个进程来执行脚本。例如，Apache 就提供了很多可集成的解释器模块。

- mod_cplusplus
- mod_perl
- mod_php
- mod_python
- mod_ruby
- mod_mono

CGI 使外部程序与 Web 服务器之间交互成为可能，不过 CGI 程序运行在独立的进程中，并对每个 Web 请求都建立一个进程。

虽然 CGI 非常容易实现，但是其效率很差且难以扩展，在处理大量请求时，进程的大量建立和消亡使操作系统性能急剧下降，而且地址空间无法共享，也限制了资源重用。

14.2 FastCGI

FastCGI（Fast Common Gateway Interface）是早期通用网关接口（CGI）的增强版本，可以让交互程序与 Web 服务器通信。

FastCGI 致力于减少网页服务器与 CGI 程序之间交互的开销，从而使服务器可以同时处理更多的网页请求。

与 CGI 为每个请求创建一个新的进程不同，FastCGI 使用持久运行的进程来处理一连串的请求，而且这些进程由 FastCGI 服务器管理（而不是 Web 服务器）。例如，Apache HTTP Server 通过 mod_fastcgi 或 mod_fcgid 模块实现了 FastCGI。

- 如果 FastCGI 进程与 Web 服务器都位于本地，当接收到一个请求时，Web 服务器把环境变量和页面请求通过一个 socket 传递给 FastCGI 进程。
- 如果 FastCGI 进程在远端的应用服务器集群中，当接收到一个请求时，Web 服务器把环境变量和页面请求通过一个 TCP connection 传递给 FastCGI 进程。

Part III

FPM

Chapter 15

Overview

15.1 PHP-FPM

FPM (FastCGI 进程管理器) 是一个可选的 PHP FastCGI 实现，并且提供了一些针对高负载网站的特性，可以用来替换 PHP FastCGI 的大部分功能，提高网站负载。

- 支持平滑停止/启动的高级进程管理功能;
- 可以工作于不同的 uid/gid/chroot 环境下，并监听不同的端口和使用不同的 php.ini 配置文件（可取代 safe_mode 的设置）;
- stdout 和 stderr 日志记录;
- 在发生意外情况的时候能够重新启动并缓存被破坏的 opcode;
- 文件上传优化支持;
- “慢日志” - 记录脚本（不仅记录文件名，还记录 PHP backtrace 信息，可以使用 ptrace 或者类似工具读取和分析远程进程的运行数据）运行所导致的异常缓慢;
- `fastcgi_finish_request()` - 在请求完成和刷新数据后继续在后台执行耗时的工作 (录入视频转换、统计处理等);
- 动态/静态子进程产生;
- 基本 SAPI 运行状态信息（类似 Apache 的 mod_status）;
- 基于 php.ini 的配置文件。

编译 PHP 时需要 `--enable-fpm` 配置选项来激活 FPM 支持，以下为 FPM 编译的具体配置参数（全部为可选参数）:

- `--with-fpm-user` 设置 FPM 运行的用户身份（默认 - nobody）
- `--with-fpm-group` 设置 FPM 运行时的用户组（默认 - nobody）
- `--with-fpm-systemd` 启用 systemd 集成 (默认 - no)
- `--with-fpm-acl` 使用 POSIX 访问控制列表 (默认 - no)

FPM 配置文件为 php-fpm.conf，其语法类似 php.ini 。

1. php-fpm.conf 全局配置段
   - `pid string`
     PID 文件的位置。默认为空。
   - `error_log string`
     错误日志的位置。默认安装路径 `#INSTALL_PREFIX#/log/php-fpm.log`。
   - `log_level string`
     错误级别。可用级别为：`alert`（必须立即处理），`error`（错误情况），`warning`（警告情况），`notice`（一般重要信息），`debug`（调试信息）。默认：`notice`。
   - `emergency_restart_threshold int`
     如果子进程在 `emergency_restart_interval` 设定的时间内收到该参数设定次数的 `SIGSEGV` 或者 `SIGBUS` 退出信息号，则 FPM 会重新启动。0 表示“关闭该功能”。默认值：0（关闭）。
   - `emergency_restart_interval mixed`
     `emergency_restart_interval` 用于设定平滑重启的间隔时间。这么做有助于解决加速器中共享内存的使用问题。可用单位：s（秒），m（分），h（小时）或者 d（天）。默认单位：s（秒）。默认值：0（关闭）。
   - `process_control_timeout mixed`
     设置子进程接受主进程复用信号的超时时间。可用单位：s（秒），m（分），h（小时）或者 d（天）。默认单位：s（秒）。默认值：0（关闭）。
   - `daemonize boolean`
     设置 FPM 在后台运行。设置“no”将 FPM 保持在前台运行用于调试。默认值：yes。
2. 运行配置区段
   在 FPM 中，可以使用不同的设置来运行多个进程池。这些设置可以针对每个进程池单独设置。
   - `listen string`
     设置接受 FastCGI 请求的地址。可用格式为：`'ip:port'`，`'port'`，`'/path/to/unix/socket'`。每个进程池都需要设置。
   - `listen.backlog int` 设置 `listen(2)` 的半连接队列长度。“-1”表示无限制。默认值：-1。
   - `listen.allowed_clients string`
     设置允许连接到 FastCGI 的服务器 IPV4 地址。等同于 PHP FastCGI (5.2.2+) 中的 `FCGI_WEB_SERVER_ADDRS` 环境变量。仅对 TCP 监听起作用。每个地址是用逗号分隔，如果没有设置或者为空，则允许任何服务器请求连接。默认值：`any`。
   - `listen.owner string`
     如果使用，表示设置 Unix 套接字的权限。在 Linux 中，读写权限必须设置，以便

用于 WEB 服务器连接。在很多 BSD 派生的系统中可以忽略权限允许自由连接。
默认值：运行所使用的用户和组，权限为 0666。

- `listen.group string`
  参见 `listen.owner`。
- `listen.mode string`
  参见 `listen.owner`。
- `user string`
  FPM 进程运行的 Unix 用户。必须设置。
- `group string`
  FPM 进程运行的 Unix 用户组。如果没有设置，则默认用户的组被使用。
- `pm string`
  设置进程管理器如何管理子进程。可用值：`static`，`ondemand`，`dynamic`。必须设置。
  - `static` - 子进程的数量是固定的（`pm.max_children`）。
  - `ondemand` - 进程在有需求时才产生(当请求时,与 `dynamic` 相反,`pm.start_servers` 在服务启动时即启动。
  - `dynamic` - 子进程的数量在下面配置的基础上动态设置：`pm.max_children`，`pm.start_servers`，`pm.min_spare_servers`，`pm.max_spare_servers`。
- `pm.max_children int`
  `pm` 设置为 `static` 时表示创建的子进程的数量，`pm` 设置为 `dynamic` 时表示最大可创建的子进程的数量。必须设置。
  该选项设置可以同时提供服务的请求数限制。类似 Apache 的 `mpm_prefork` 中 `MaxClients` 的设置和普通 PHP FastCGI 中的 `PHP_FCGI_CHILDREN` 环境变量。
- `pm.start_servers in`
  设置启动时创建的子进程数目。仅在 `pm` 设置为 `dynamic` 时使用。默认值:`min_spare_servers + (max_spare_servers - min_spare_servers) / 2`。
- `pm.min_spare_servers int`
  设置空闲服务进程的最低数目。仅在 `pm` 设置为 `dynamic` 时使用。必须设置。
- `pm.max_spare_servers int`
  设置空闲服务进程的最大数目。仅在 `pm` 设置为 `dynamic` 时使用。必须设置。
- `pm.max_requests int`
  设置每个子进程重生之前服务的请求数。对于可能存在内存泄漏的第三方模块来说是非常有用的。如果设置为'0'则一直接受请求,等同于 `PHP_FCGI_MAX_REQUESTS` 环境变量。默认值：0。
- `pm.status_path string`
  FPM 状态页面的网址。如果没有设置，则无法访问状态页面，默认值：无。

- `ping.path string`
  FPM 监控页面的 ping 网址。如果没有设置，则无法访问 ping 页面。该页面用于外部检测 FPM 是否存活并且可以响应请求。请注意必须以斜线开头（/）。
- `ping.response string`
  用于定义 ping 请求的返回响应。返回为 HTTP 200 的 text/plain 格式文本。默认值：`pong`。
- `request_terminate_timeout mixed`
  设置单个请求的超时中止时间。该选项可能会对 php.ini 设置中的 `'max_execution_time'` 因为某些特殊原因没有中止运行的脚本有用。设置为 `'0'` 表示 `'Off'`。可用单位：s（秒），m（分），h（小时）或者 d（天）。默认单位：s（秒）。默认值：0（关闭）。
- `request_slowlog_timeout mixed`
  当一个请求该设置的超时时间后，就会将对应的 PHP 调用堆栈信息完整写入到慢日志中。设置为 `'0'` 表示 `'Off'`。可用单位：s（秒），m（分），h（小时）或者 d（天）。默认单位：s（秒）。默认值：0（关闭）。
- `slowlog string`
  慢请求的记录日志。默认值：`#INSTALL_PREFIX#/log/php-fpm.log.slow`。
- `rlimit_files int`
  设置文件打开描述符的 rlimit 限制。默认值：系统定义值。
- `rlimit_core int`
  设置核心 `rlimit` 最大限制值。可用值：`'unlimited'`，0 或者正整数。默认值：系统定义值。
- `chroot string`
  启动时的 Chroot 目录。所定义的目录需要是绝对路径。如果没有设置，则 chroot 不被使用。
- `chdir string`
  设置启动目录，启动时会自动 Chdir 到该目录。所定义的目录需要是绝对路径。默认值：当前目录，或者根目录（chroot 时）。
- `catch_workers_output boolean`
  重定向运行过程中的 `stdout` 和 `stderr` 到主要的错误日志文件中。如果没有设置，`stdout` 和 `stderr` 将会根据 FastCGI 的规则被重定向到`/dev/null`。默认值：无。

还可以在为一个运行池传递附加的环境变量，或者更新 PHP 的配置值。可以在 php-fpm.conf 中如下面的配置参数来做到：

```
env[HOSTNAME] = $HOSTNAME
      env[PATH] = /usr/local/bin:/usr/bin:/bin
      env[TMP] = /tmp
```

```
    env[TMPDIR] = /tmp
    env[TEMP] = /tmp

    php_admin_value[sendmail_path] = /usr/sbin/sendmail -t -i -f www@my.domain.com
    php_flag[display_errors] = off
    php_admin_value[error_log] = /var/log/fpm-php.www.log
    php_admin_flag[log_errors] = on
    php_admin_value[memory_limit] = 32M
```

PHP 配置值通过 `php_value` 或者 `php_flag` 设置，并且会覆盖以前的值。注意 `disable_functions` 或者 `disable_classes` 在 php.ini 之中定义的值不会被覆盖掉，但是会将新的设置附加在原有值的后面。

使用 `php_admin_value` 或者 `php_admin_flag` 定义的值，不能被 PHP 代码中的 `ini_set()` 覆盖。

自 PHP 5.3.3 起，也可以通过 Web 服务器设置 PHP 的设定。例如，在 nginx.conf 中设定 PHP 的示例如下：

```
set $php_value "pcre.backtrack_limit=424242";
set $php_value "$php_value \n pcre.recursion_limit=99999";
fastcgi_param PHP_VALUE $php_value;

fastcgi_param PHP_ADMIN_VALUE "open_basedir=/var/www/htdocs";
```

上述这些设定是以 FastCGI 标头传递给 php-fpm，因此 php-fpm 不应被绑定到外部网可以访问的地址上，否则任何人就能可以修改 PHP 的配置选项。

Part IV

Foundation

Chapter 16

PHP Syntax

PHP 标识符用于赋予变量、常量、函数、类或方法的名称，只能由字母（所有英文字符以及 ASCII 码值在 127 ～ 255 的所有字符)、数字或下划线组成。

标识符只能以字母或下划线开头，并且 PHP 自身定义的关键字不能作为常量、函数名或类名，但是可以将它们作为变量使用。

16.1 PHP tags

PHP Web 页面和通常的 HTML 页面一样处理，可以用通常建立 HTML 页面的方法来建立和编辑它们，但是用户无法在浏览器中通过查看源文档的方式来查看 PHP 的源代码 - 而是只能看到 PHP 文件的输出——即纯粹的 HTML。这是因为在结果返回浏览器之前，脚本就已经在服务器执行了。

当解析一个 PHP 文件时，PHP 会寻找起始和结束标记，也就是 `<?php` 和 `?>` ，这告诉 PHP 开始和停止解析二者之间的代码，因此 PHP 可以被嵌入到各种不同的文档中去，任何起始和结束标记之外的部分都会被 PHP 解析器忽略。

PHP 的脚本块[1]以 `<?php` 开始，以 `?>` 结束，可以把 PHP 的脚本块放置在文档中的任何位置。

当然，在支持简写的服务器上，可以使用短标记 `<?` 和 `?>` 来开始和结束脚本块, 但不鼓励使用。只有通过激活 php.ini 中的 short_open_tag 配置指令或者在编译 PHP 时使用了配置选项 `--enable-short-tags` 时才能使用短标记。

为了达到最好的兼容性，推荐使用标准形式 (`<?php ?>`)，而不是简写形式。

```
<?php
```

[1]用 `<?php` 来表示 PHP 标识符的起始，然后放入 PHP 语句并通过加上一个终止标识符 `?>` 来退出 PHP 模式，可以根据需要在 HTML 文件中开启或关闭 PHP 模式。

```
  ...
?>
```

如果文件内容是纯 PHP 代码，最好在文件末尾删除 PHP 结束标记。这可以避免在 PHP 结束标记之后万一意外加入了空格或者换行符，会导致 PHP 开始输出这些空白，而脚本中此时并无输出的意图。

```
<?php
echo "Hello world";

// ... more code

echo "Last statement";

// 脚本至此结束，而且并无 PHP 结束标记
```

凡是在一对开始和结束标记之外的内容都会被 PHP 解析器忽略，这使得 PHP 文件可以具备混合内容，因此 PHP 文件通常会包含 HTML 标签，就像一个 HTML 文件，以及一些 PHP 脚本代码。

在 Web 服务器根目录（DOCUMENT_ROOT）下建立一个文件名为 hello.php，然后完成如下内容，它可以向浏览器输出文本”Hello World”：

```
<!DOCTYPE html>
<html>
<head>
  <title>PHP Example</title>
</head>
<body>
<?php
  echo "Hello World";
?>
</body>
</html>
```

在浏览器的地址栏里输入 Web 服务器的 URL 访问这个文件，在结尾加上“/hello.php”。如果本地开发，那么这个 URL 一般是 http://localhost/hello.php 或者 http://127.0.0.1/hello.php，当然这取决于 Web 服务器的设置。如果所有的设置都正确，那么这个文件将被 PHP 解析，浏览器中将会输出如下结果：

```
<!DOCTYPE html>
<html>
 <head>
  <title>PHP Example</title>
```

```
</head>
<body>
<p>Hello World</p>
</body>
</html>
```

该程序非常的简单，它仅仅只是利用了 PHP 的 echo 语句显示了 Hello World。注意，这个范例和其它用 C 或 Perl 语言写的脚本之间的区别 ——与用大量的命令来编写程序以输出 HTML 不同的是，PHP 页面就是 HTML，只不过在其中嵌入了一些代码来做一些事情。

有两种通过 PHP 来输出文本的基础指令：echo 和 print。在上面的例子中就是使用 echo 语句来输出文本”Hello World”。

下面建立一个最著名的 PHP 脚本，调用函数 `phpinfo()`，将会看到很多有关自己系统的有用信息，例如预定义变量、已经加载的 PHP 模块和配置信息。

```
<!DOCTYPE html>
<html>
<head>
  <title>PHP Example</title>
</head>
<body>
<?php
  phpinfo();
?>
</body>
</html>
```

尽管换行在 HTML 中的实际意义不是很大，但适当地使用换行可以使 HTML 代码易读且美观。PHP 会在输出时自动删除其结束符 `?>` 后的一个换行。该功能主要是针对在一个页面中嵌入多段 PHP 代码或者包含了无实质性输出的 PHP 文件而设计，与此同时也造成了一些疑惑。如果需要在 PHP 结束符 `?>` 之后输出换行的话，可以在其后加一个空格，或者在最后的一个 `echo/print` 语句中加入一个换行。

16.2 PHP Mode

当 PHP 解释器遇到`?>` 结束标记时就简单地将其后内容原样输出（除非马上紧接换行）直到碰到下一个开始标记，要输出大段文本时，跳出 PHP 解析模式通常比将文本通过 echo 或 print 输出更有效率。

例外是处于条件语句中间时，此时 PHP 解释器会根据条件判断来决定哪些输出，哪些跳过。

```
<?php if ($expression == true): ?>
```

```
 This will show if the expression is true.
<?php else: ?>
 Otherwise this will show.
<?php endif; ?>
```

上例中 PHP 将跳过条件语句未达成的段落，即使该段落位于 PHP 开始和结束标记之外。由于 PHP 解释器会在条件未达成时直接跳过该段条件语句块，因此 PHP 会根据条件来忽略之。

此外注意如果将 PHP 嵌入到 XML 或 XHTML 中则需要使用 `<?php ?>` 标记以保持符合标准[2]。

PHP 的开始和结束标记中 `<?php ?>` 和 `<script language="php"> </script>` 总是可用的。另两种是短标记和 ASP 风格标记，可以在 php.ini 配置文件中打开或关闭。

在以下情况应避免使用短标记：开发需要再次发布的程序或者库，或者在用户不能控制的服务器上开发。因为目标服务器可能不支持短标记。为了代码的移植及发行，确保不要使用短标记。尽管有些人觉得短标记和 ASP 风格标记很方便，但移植性较差，通常不推荐使用。

```
1. <?php echo 'if you want to serve XHTML or XML documents, do it like this'; ?>

2. <script language="php">
       echo 'some editors (like FrontPage) don\'t like processing instructions';
   </script>

3. <? echo 'this is the simplest, an SGML processing instruction'; ?>
   <?= expression ?> This is a shortcut for "<? echo expression ?>"

4. <% echo 'You may optionally use ASP-style tags'; %>
   <%= $variable; # This is a shortcut for "<% echo . . ." %>
```

上例中的 1 和 2 中使用的标记总是可用的，其中示例 1 中是最常用，并建议使用的。

短标记（上例 3）仅在通过 php.ini 配置文件中的指令 short_open_tag 打开后才可用[3]，或者在 PHP 编译时加入了 `--enable-short-tags` 选项，而 ASP 风格标记（上例 4）仅在通过 php.ini 配置文件中的指令 asp_tags 打开后才可用。

Web 服务器中的 PHP 模块设置好以后，而且通过浏览器访问的 URL 确实指向了服务器上的这个文件，那么 PHP 文件无需被执行或以任何方式指定，服务器会找到该文件并提供给 PHP 进行解释，因为使用了“.php”的扩展名，服务器已被配置成自动传递有着“.php”扩展名的文件给 PHP，但如果只是从本地文件系统调用这个文件，它不会被 PHP 解析。

[2]在 PHP 5.2 和之前的版本中，解释器不允许一个文件的全部内容就是一个开始标记 `<?php`。自 PHP 5.3 起则允许此种文件，但要开始标记后有一个或更多白空格符

[3]自 PHP 5.4 起，短格式的 echo 标记 `<?=` 总会被识别并且合法，而不管 short_open_tag 的设置是什么。

16.3 PHP Separations

同 C 或 Perl 一样，PHP 中的每个代码行都必须以分号[4]结束。

一段 PHP 代码中的结束标记隐含表示了一个分号；在一个 PHP 代码段中的最后一行可以不用分号结束。如果后面还有新行，则代码段的结束标记包含了行结束。

```
<?php
   echo "This is a test";
?>

<?php echo "This is a test" ?>

<?php echo 'We omitted the last closing tag';
```

文件末尾的 PHP 代码段结束标记可以不要，有些情况下当使用 include 或者 require 时省略掉会更好些，这样不期望的空白符就不会出现在文件末尾，之后仍然可以输出响应标头。在使用输出缓冲时也很便利，就不会看到由包含文件生成的不期望的空白符。

16.4 PHP Comments

PHP 支持 C/C++/C#/Java 和 Unix Shell 风格（Perl 风格）的注释，PHP 使用 `//` 来编写单行注释，或者使用 `/*` 和 `*/` 来编写大的注释块。

```
<?php
   echo "This is a test"; // This is a one-line c++ style comment
   /* This is a multi line comment
      yet another line of comment */
   echo "This is yet another test";
   echo 'One Final Test'; # This is a one-line shell-style comment
?>
```

单行注释仅仅注释到行末或者当前的 PHP 代码块，视乎哪个首先出现。这意味着在 `// ... ?>` 或者 `# ... ?>` 之后的 HTML 代码将被显示出来：`?>` 跳出了 PHP 模式并返回了 HTML 模式，`//` 或 `#` 并不能影响到这一点。如果启用了 asp_tags 配置选项，其行为和 `// %>` 或 `# %>` 相同。不过，`</script>` 标记在单行注释中不会跳出 PHP 模式。

```
<h1>This is an <?php # echo 'simple';?> example</h1>
<p>The header above will say 'This is an example'.</p>
```

C 风格的注释在碰到第一个 `*/` 时结束，因此要确保不要嵌套 C 风格的注释。试图注释掉一大块代码时很容易出现该错误。

[4]分号是一种分隔符，用于把指令集区分开来。

```
<?php
 /*
   echo "This is a test"; /* This comment will cause a problem */
 */
?>
```

和客户端的 JavaScript 不同的是，PHP 代码是运行在服务端的。如果在服务器上建立了如上例类似的代码，则在运行该脚本后，客户端就能接收到其结果，但他们无法得知其背后的代码是如何运作的。甚至可以将 web 服务器设置成让 PHP 来处理所有的 HTML 文件，这么一来，用户就无法得知服务端到底做了什么。

如果希望用文本编辑工具[5]来处理 PHP 脚本，必须保证将结果存成了纯文本格式，否则 PHP 将无法读取并运行这些脚本。

[5]如果使用 Windows 记事本来编写 PHP 脚本，需要注意在保存文件时，文件的后缀名应该为.php（记事本将自动在文件名后面加上.txt 后缀，除非采取以下措施之一来避免这种情况）。当保存文件时，系统会提示指定文件的文件名，这时需要将文件名加上引号（例如 “hello.php”）。或者，也可以点击“另存为”对话框中的“保存类型”下拉菜单，并将设置改为“所有文件”，这样在输入文件名的时候就不用加引号了。

Chapter 17

PHP Type

17.1 Overview

PHP 支持 8 种原始数据类型，其中首先是以下四种标量类型：

- 整型（integer）
- 浮点型，也称作 double（float）[1]
- 布尔型（boolean）
- 字符串（string）

两种复合类型：

- 数组（array）
- 对象（object）

两种特殊类型：

- NULL（无类型）
- 资源（resource）

为了确保代码的易读性，还可以引入一些伪类型以及伪变量如下：

- mixed（混合类型）
- number（数字类型）
- callback（回调类型）
- $...（伪变量）

确切地说，变量的类型是由 PHP 根据该变量使用的上下文在运行时决定的，并且根据其当时的类型在特定场合下会表现出不同的值，可以使用 var_dump() 函数查看某个表达式的值和类型。

- 如果只是想得到一个易读懂的类型的表达方式用于调试，用 gettype() 函数。

[1]实际上 double 和 float 是相同的，由于一些历史的原因，这两个名称同时存在。

- 如果要将一个变量强制转换为某类型，可以对其使用强制转换或者 settype() 函数。

17.1.1 gettype()

17.1.2 settype()

17.1.3 is_type

要查看某个类型，不要用 gettype()，而用 is_type() 函数。

```
<?php
$a_bool = TRUE; // a boolean
$a_str = "foo"; // a string
$a_str2 = 'foo'; // a string
$an_int = 12;  // an integer

echo gettype($a_bool); // prints out: boolean
echo gettype($a_str); // prints out: string

// If this is an integer, increment it by four
if (is_int($an_int)) {
    $an_int += 4;
}

// If $bool is a string, print it out
// (does not print out anything)
if (is_string($a_bool)) {
    echo "String: $a_bool";
}
?>
```

17.2 Boolean

boolean 表达了真值，可以为 TRUE 或 FALSE。
要指定一个布尔值，使用关键字 TRUE 或 FALSE，两个都不区分大小写。

```
<?php
$foo = True; // assign the value TRUE to $foo
?>
```

通常运算符所返回的 boolean 值结果会被传递给控制流程。

```php
<?php
// == 是一个操作符，它检测两个变量是否相等，并返回一个布尔值
if ($action == "show_version") {
    echo "The version is 1.23";
}

// 这样做是不必要的...
if ($show_separators == TRUE) {
    echo "<hr>\n";
}

// ...因为可以使用下面这种简单的方式：
if ($show_separators) {
    echo "<hr>\n";
}
?>
```

17.2.1 Casting

要明确地将一个值转换成 boolean，用 (bool) 或者 (boolean) 来强制转换。

实际上很多情况下不需要用强制转换。当运算符、函数或者流程控制结构需要一个 boolean 参数时，该值就会被自动转换。例如，当转换为 boolean 时，以下值被认为是 FALSE：

- 布尔值 FALSE 本身
- 整型值 0（零）
- 浮点型值 0.0（零）
- 空字符串，以及字符串"0"
- 不包括任何元素的数组
- 不包括任何成员变量的对象（仅 PHP 4.0 适用）
- 特殊类型 NULL（包括尚未赋值的变量）
- 从没有任何标记（tags）的 XML 文档生成的 SimpleXML 对象

所有其它值[2]都被认为是 TRUE（包括任何资源类型）。

```php
<?php
var_dump((bool) "");    // bool(false)
var_dump((bool) 1);     // bool(true)
var_dump((bool) -2);    // bool(true)
var_dump((bool) "foo"); // bool(true)
var_dump((bool) 2.3e5); // bool(true)
```

[2] -1 和其它非零值（不论正负）一样，被认为是 TRUE。

```php
var_dump((bool) array(12)); // bool(true)
var_dump((bool) array()); // bool(false)
var_dump((bool) "false"); // bool(true)
?>
```

17.3 Integer

一个 integer 是集合 $\mathbb{Z} = \{..., -2, -1, 0, 1, 2, ...\}$ 中的一个数。

整型值可以使用十进制，十六进制，八进制或二进制表示，前面可以加上可选的符号（-或者 +）。

- 要使用八进制表达，数字前必须加上 0（零）。
- 要使用十六进制表达，数字前必须加上 0x。
- 要使用二进制表达，数字前必须加上 0b。

```php
<?php
$a = 1234; // 十进制数
$a = -123; // 负数
$a = 0123; // 八进制数（等于十进制 83）
$a = 0x1A; // 十六进制数（等于十进制 26）
?>
```

整型 integer 的正式描述为：

```
decimal     : [1-9][0-9]*
            | 0

hexadecimal : 0[xX][0-9a-fA-F]+

octal       : 0[0-7]+

binary      : 0b[01]+

integer     : [+-]?decimal
            | [+-]?hexadecimal
            | [+-]?octal
            | [+-]?binary
```

整型数的字长和平台有关，尽管通常最大值是大约二十亿（32 位有符号）。64 位平台下的最大值通常是大约 9E18。PHP 不支持无符号整数。Integer 值的字长可以用常量 PHP_INT_SIZE 来表示，最大值可以用常量 PHP_INT_MAX 来表示。

- PHP_INT_SIZE
- PHP_INT_MAX

如果向八进制数传递了一个非法数字（即 8 或 9），则后面其余数字会被忽略。

```
<?php
var_dump(01090); // 八进制 010 = 十进制 8
?>
```

17.3.1 Overflow

如果给定的一个数超出了 integer 的范围，将会被解释为 float。同样，如果执行的运算结果超出了 integer 范围，也会返回 float。

32 位系统下的整数溢出示例如下：

```
<?php
$large_number = 2147483647;
var_dump($large_number);                    // int(2147483647)

$large_number = 2147483648;
var_dump($large_number);                    // float(2147483648)

$million = 1000000;
$large_number = 50000 * $million;
var_dump($large_number);                    // float(50000000000)
?>
```

64 位系统下的整数溢出示例如下：

```
<?php
$large_number = 9223372036854775807;
var_dump($large_number);                    // int(9223372036854775807)

$large_number = 9223372036854775808;
var_dump($large_number);                    // float(9.2233720368548E+18)

$million = 1000000;
$large_number = 50000000000000 * $million;
var_dump($large_number);                    // float(5.0E+19)
?>
```

PHP 中没有整除的运算符。1/2 产生出 float 0.5。值可以舍弃小数部分强制转换为 integer，或者使用 round() 函数可以更好地进行四舍五入。

```
<?php
var_dump(25/7);      // float(3.5714285714286)
var_dump((int) (25/7)); // int(3)
var_dump(round(25/7)); // float(4)
?>
```

17.3.2 Casting

要明确地将一个值转换为 integer，用 (int) 或 (integer) 强制转换。

大多数情况下都不需要强制转换，因为当运算符、函数或流程控制需要一个 integer 参数时，值会自动转换，而且还可以通过函数 intval() 来将一个值转换成整型。

- 从布尔值转换

  FALSE 将产生出 0（零），TRUE 将产生出 1（壹）。
- 从浮点型转换

  当从浮点数转换成整数时，将向下取整。
- 从字符串转换
- 从其它类型转换

  没有定义从其它类型转换为整型的行为。不要依赖任何现有的行为，因为它会未加通知地改变。

如果浮点数超出了整数范围（32 位平台下通常为 +/- 2.15e+9 = 2^31，64 位平台下通常为 +/- 9.22e+18 = 2^63），则结果为未定义，因为没有足够的精度给出一个确切的整数结果。在此情况下没有警告，甚至没有任何通知。

另外，一定不要将未知的分数强制转换为 integer，这样有时会导致不可预料的结果。

```
<?php
echo (int) ( (0.1+0.7) * 10 ); // 显示 7!
?>
```

17.4 Float

浮点型（也叫浮点数 float，双精度数 double 或实数 real）可以用以下任一语法定义：

```
<?php
$a = 1.234;
$b = 1.2e3;
$c = 7E-10;
?>
```

浮点数的形式表示：

```
LNUM            [0-9]+
DNUM            ([0-9]*[\.]{LNUM}) | ({LNUM}[\.][0-9]*)
EXPONENT_DNUM [+-]?(({LNUM} | {DNUM}) [eE][+-]? {LNUM})
```

浮点数的字长和平台相关，尽管通常最大值是 1.8e308 并具有 14 位十进制数字的精度（64 位 IEEE 格式）。

17.4.1 Precision

浮点数的精度有限，实际上尽管取决于系统，PHP 通常使用 IEEE 754 双精度格式。

- IEEE 754 双精度格式由取整而导致的最大相对误差为 1.11e-16。
- 非基本数学运算可能会给出更大误差，并且要考虑到进行复合运算时的误差传递。

此外，以十进制能够精确表示的有理数如 0.1 或 0.7，无论有多少尾数都不能被内部所使用的二进制精确表示，因此不能在不丢失一点点精度的情况下转换为二进制的格式，这样就会造成混乱的结果。例如，floor((0.1+0.7)*10) 通常会返回 7 而不是预期中的 8，因为该结果内部的表示其实是类似 7.9999999999999991118...。

永远不要相信浮点数结果精确到了最后一位，也永远不要比较两个浮点数是否相等。如果确实需要更高的精度，应该使用任意精度数学函数或者 gmp 函数。

由于内部表达方式的原因，在 PHP 中比较两个浮点数是否相等是有问题的，不过还是有迂回的方法来比较浮点数值的。

要测试浮点数是否相等，要使用一个仅比该数值大一丁点的最小误差值，该值也被称为机器极小值（epsilon）或最小单元取整数，也是计算中所能接受的最小的差别值。

$a 和 $b 在小数点后五位精度内都是相等的。

```php
<?php
$a = 1.23456789;
$b = 1.23456780;
$epsilon = 0.00001;

if(abs($a-$b) < $epsilon) {
    echo "true";
}
?>
```

17.4.2 Casting

在将其他类型转换为浮点数类型时，情况类似于先将值转换成整型，然后再转换成浮点类型，但是字符串类型除外。

17.4.3 NaN

某些数学运算会产生一个由常量 NAN 所代表的结果。此结果代表着一个在浮点数运算中未定义或不可表述的值。任何拿此值与其它任何值进行的松散或严格比较的结果都是 FALSE。

由于 NAN 代表着任何不同值，不应拿 NAN 去和其它值进行比较（包括其自身），应该用 is_nan() 来检查。

17.5 String

一个字符串 string 就是由一系列的字符组成，其中每个字符等同于一个字节。这意味着 PHP 只能支持 256 的字符集，因此 PHP 不支持 Unicode，string 最大可以达到 2GB。

一个字符串可以用 4 种方式表达：

- 单引号
- 双引号
- heredoc
- nowdoc

17.5.1 single quotation

定义一个字符串的最简单的方法是用单引号把它包围起来（字符 '）。

- 要表达一个单引号自身，需在它的前面加个反斜线（\）来转义。
- 要表达一个反斜线自身，则用两个反斜线（\\）。

其它任何方式的反斜线都会被当成反斜线本身。也就是说，如果想使用其它转义序列例如 \r 或者 \n，并不代表任何特殊含义，就单纯是这两个字符本身。

不像双引号和 heredoc 语法结构，在单引号字符串中的变量和特殊字符的转义序列将不会被替换。

```
<?php
echo 'this is a simple string';

// 可以录入多行
echo 'You can also have embedded newlines in
strings this way as it is
okay to do';

// 输出： Arnold once said: "I'll be back"
echo 'Arnold once said: "I\'ll be back"';
```

```
// 输出： You deleted C:\*.*?
echo 'You deleted C:\\*.*?';

// 输出： You deleted C:\*.*?
echo 'You deleted C:\*.*?';

// 输出： This will not expand: \n a newline
echo 'This will not expand: \n a newline';

// 输出： Variables do not $expand $either
echo 'Variables do not $expand $either';
?>
```

17.5.2 double quotation

用双引号定义的字符串最重要的特征是变量会被解析。

如果字符串是包围在双引号（"）中，PHP 将对一些特殊的字符进行解析[3]。

表 17.1: 转义字符

序列	含义
\n	换行（ASCII 字符集中的 LF 或 0x0A (10)）
\r	回车（ASCII 字符集中的 CR 或 0x0D (13)）
\t	水平制表符（ASCII 字符集中的 HT 或 0x09 (9)）
\v	垂直制表符（ASCII 字符集中的 VT 或 0x0B (11)）（自 PHP 5.2.5 起）
\e	Escape（ASCII 字符集中的 ESC 或 0x1B (27)）（自 PHP 5.4.0 起）
\f	换页（ASCII 字符集中的 FF 或 0x0C (12)）（自 PHP 5.2.5 起）
\\	反斜线
\$	美元标记
\"	双引号
\[0-7]{1,3}	符合该正则表达式序列的是一个以八进制方式来表达的字符
\x[0-9A-Fa-f]{1,2}	符合该正则表达式序列的是一个以十六进制方式来表达的字符

具体来说，转义字符是“\”和其他字符合起来表示的一个特殊字符，通常是一些非打印字符。

[3]和单引号字符串一样，转义任何其它字符都会导致反斜线被显示出来。

17.5.3 heredoc

第三种表达字符串的方法是用 heredoc 句法结构：<<<。在该运算符之后要提供一个标识符，然后换行。接下来是字符串 string 本身，最后要用前面定义的标识符作为结束标志。

结束时所引用的标识符必须在该行的第一列，而且标识符的命名也要像其它标签一样遵守 PHP 的规则：只能包含字母、数字和下划线，并且必须以字母和下划线作为开头。

要注意的是结束标识符这行除了可能有一个分号（;）外，绝对不能包含其它字符。这意味着标识符不能缩进，分号的前后也不能有任何空白或制表符。更重要的是结束标识符的前面必须是个被本地操作系统认可的换行，比如在 UNIX 和 Mac OS X 系统中是 \n，而结束定界符（可能其后有个分号）之后也必须紧跟一个换行。

```
<?php
class foo {
    public $bar = <<<EOT // Attention: Illegal
bar
    EOT;
}
?>
```

如果不遵守该规则导致结束标识不“干净”，PHP 将认为它不是结束标识符而继续寻找。如果在文件结束前也没有找到一个正确的结束标识符，PHP 将会在最后一行产生一个解析错误。

Heredocs 结构不能用来初始化类的属性，该限制仅对 heredoc 包含变量时有效。

Heredoc 结构就象是没有使用双引号的双引号字符串，这就是说在 heredoc 结构中单引号不用被转义，但是上文中列出的转义序列还可以使用。变量将被替换，但在 heredoc 结构中含有复杂的变量时要格外小心。

```
<?php
$str = <<<EOD
Example of string
spanning multiple lines
using heredoc syntax.
EOD;

/* 含有变量的更复杂示例 */
class foo
{
    var $foo;
    var $bar;

    function foo()
```

```
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<EOT
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should print a capital 'A': \x41
EOT;
?>
```

以上例程会输出：

```
My name is "MyName". I am printing some Foo.
Now, I am printing some Bar2.
This should print a capital 'A': A
```

- 也可以把 Heredoc 结构用在函数参数中来传递数据：

```
<?php
var_dump(array(<<<EOD
foobar!
EOD
));
?>
```

- 也可以用 Heredoc 结构来初始化静态值：

```
<?php
// 静态变量
function foo()
{
    static $bar = <<<LABEL
Nothing in here...
LABEL;
}

// 类的常量、属性
```

```
class foo
{
    const BAR = <<<FOOBAR
Constant example
FOOBAR;

    public $baz = <<<FOOBAR
Property example
FOOBAR;
}
?>
```

- 还可以在 Heredoc 结构中用双引号来声明标识符：

```
<?php
echo <<<"FOOBAR"
Hello World!
FOOBAR;
?>
```

17.5.4 nowdoc

- heredoc 结构类似于双引号字符串；
- nowdoc 结构是类似于单引号字符串的。

Nowdoc 结构很象 heredoc 结构，但是 nowdoc 中不进行解析操作，这种结构很适合用于嵌入 PHP 代码或其它大段文本而无需对其中的特殊字符进行转义。

与 SGML 的 `<![CDATA[ ]]>` 结构是用来声明大段的不用解析的文本类似，nowdoc 结构也有相同的特征。

一个 nowdoc 结构也用和 heredocs 结构一样的标记 <<<，但是跟在后面的标识符要用单引号括起来，即 `<<<'EOT'`。

Heredoc 结构的所有规则也同样适用于 nowdoc 结构，尤其是结束标识符的规则。

```
<?php
$str = <<<'EOD'
Example of string
spanning multiple lines
using nowdoc syntax.
EOD;

/* 含有变量的更复杂的示例 */
class foo
```

```
{
    public $foo;
    public $bar;

    function foo()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$name = 'MyName';

echo <<<'EOT'
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
EOT;
?>
```

以上例程会输出：

```
My name is "$name". I am printing some $foo->foo.
Now, I am printing some {$foo->bar[1]}.
This should not print a capital 'A': \x41
```

和 heredoc 结构不同，nowdoc 结构可以用在任意的静态数据环境中，最典型的示例是用来初始化类的属性或常量：

```
<?php
class foo {
    public $bar = <<<'EOT'
bar
EOT;
}
?>
```

17.5.5 Variable Parsing

当字符串用双引号或 heredoc 结构定义时，其中的变量将会被解析。

这里共有两种语法规则：一种简单规则，一种复杂规则。简单的语法规则是最常用和最方便的，它可以用最少的代码在一个 string 中嵌入一个变量，一个 array 的值，或一个 object 的属性。

复杂规则语法的显著标记是用花括号包围的表达式。

- 简单语法

  当 PHP 解析器遇到一个美元符号（$）时，它会和其它很多解析器一样去组合尽量多的标识以形成一个合法的变量名，可以用花括号来明确变量名的界线。

```
<?php
$juice = "apple";

echo "He drank some $juice juice.".PHP_EOL;
// Invalid. "s" is a valid character for a variable name, but the variable is
    $juice.
echo "He drank some juice made of $juices.";
?>
```

  类似的，一个 array 索引或一个 object 属性也可被解析。数组索引要用方括号（]）来表示索引结束的边际，对象属性则是和上述的变量规则相同。

```
<?php
$juices = array("apple", "orange", "koolaid1" => "purple");

echo "He drank some $juices[0] juice.".PHP_EOL;
echo "He drank some $juices[1] juice.".PHP_EOL;
echo "He drank some juice made of $juice[0]s.".PHP_EOL; // Won't work
echo "He drank some $juices[koolaid1] juice.".PHP_EOL;

class people {
    public $john = "John Smith";
    public $jane = "Jane Smith";
    public $robert = "Robert Paulsen";

    public $smith = "Smith";
}

$people = new people();

echo "$people->john drank some $juices[0] juice.".PHP_EOL;
echo "$people->john then said hello to $people->jane.".PHP_EOL;
echo "$people->john's wife greeted $people->robert.".PHP_EOL;
echo "$people->robert greeted the two $people->smiths."; // Won't work
```

```
?>
```

如果想要表达更复杂的结构，请用复杂语法。

- 复杂（花括号）语法
  复杂语法不是因为其语法复杂而得名，而是因为它可以使用复杂的表达式。
  任何具有 string 表达的标量变量，数组单元或对象属性都可使用此语法，只需简单地像在 string 以外的地方那样写出表达式，然后用花括号 { 和 } 把它括起来即可。
  由于 { 无法被转义，只有 $ 紧挨着 { 时才会被识别，可以用 {\$ 来表达 {$，下面的示例可以更好的解释：

```
<?php
// 显示所有错误
error_reporting(E_ALL);

$great = 'fantastic';

// 无效，输出：This is { fantastic}
echo "This is { $great}";

// 有效，输出： This is fantastic
echo "This is {$great}";
echo "This is ${great}";

// 有效
echo "This square is {$square->width}00 centimeters broad.";

// 有效，只有通过花括号语法才能正确解析带引号的键名
echo "This works: {$arr['key']}";

// 有效
echo "This works: {$arr[4][3]}";

// 这是错误的表达式，因为就象 $foo[bar] 的格式在字符串以外也是错的一样。
// 换句话说，只有在 PHP 能找到常量 foo 的前提下才会正常工作；这里会产生一个
// E_NOTICE (undefined constant) 级别的错误。
echo "This is wrong: {$arr[foo][3]}";

// 有效，当在字符串中使用多重数组时，一定要用括号将它括起来
echo "This works: {$arr['foo'][3]}";

// 有效
```

```
echo "This works: " . $arr['foo'][3];

echo "This works too: {$obj->values[3]->name}";

echo "This is the value of the var named $name: {${$name}}";

echo "This is the value of the var named by the return value of getName(): {${
    getName()}}";

echo "This is the value of the var named by the return value of \$object->getName
    (): {${$object->getName()}}";

// 无效，输出： This is the return value of getName(): {getName()}
echo "This is the return value of getName(): {getName()}";
?>
```

也可以在字符串中用此语法通过变量来调用类的属性。

```
<?php
class foo {
    var $bar = 'I am bar.';
}

$foo = new foo();
$bar = 'bar';
$baz = array('foo', 'bar', 'baz', 'quux');
echo "{$foo->$bar}\n";
echo "{$foo->$baz[1]}\n";
?>
```

函数、方法、静态类变量和类常量只有在 PHP 5 以后才可在 {$} 中使用。

- 只有在该字符串被定义的命名空间中才可以将其值作为变量名来访问。
- 只单一使用花括号 ({}) 无法处理从函数或方法的返回值或者类常量以及类静态变量的值。

```
<?php
// 显示所有错误
error_reporting(E_ALL);

class beers {
    const softdrink = 'rootbeer';
    public static $ale = 'ipa';
}
```

```
$rootbeer = 'A & W';
$ipa = 'Alexander Keith\'s';

// 有效，输出： I'd like an A & W
echo "I'd like an {${beers::softdrink}}\n";

// 也有效，输出： I'd like an Alexander Keith's
echo "I'd like an {${beers::$ale}}\n";
?>
```

17.5.6 String Operation

- string 中的字符可以通过一个从 0 开始的下标（比如 $str[42]），用类似 array 结构中的方括号包含对应的数字来访问和修改，因此可以把 string 当成字符组成的 array。
- 函数 substr() 和 substr_replace() 可用于操作多于一个字符的情况。
- 同样地，string 也可用花括号访问（比如 $str{42}）。

PHP 的字符串在内部是字节组成的数组，因此用花括号访问或修改字符串对多字节字符集很不安全，实际上仅应对单字节编码（例如 ISO-8859-1）的字符串进行上述此类操作。

用 [] 或 {} 访问任何其它类型（不包括数组或具有相应接口的对象实现）的变量只会返回 NULL，从 PHP 5.5 开始增加了直接在字符串原型中用 [] 或 {} 访问字符的支持。

- 用超出字符串长度的下标写入将会拉长该字符串并以空格填充。
- 非整数类型下标会被转换成整数。
- 非法下标类型会产生一个 E_NOTICE 级别错误。
- 用负数下标写入字符串时会产生一个 E_NOTICE 级别错误。
- 用负数下标读取字符串时返回空字符串。
- 写入时只用到了赋值字符串的第一个字符。
- 用空字符串赋值则赋给的值是 NULL 字符。

```
<?php
// 取得字符串的第一个字符
$str = 'This is a test.';
$first = $str[0];

// 取得字符串的第三个字符
$third = $str[2];

// 取得字符串的最后一个字符
$str = 'This is still a test.';
```

```
$last = $str[strlen($str)-1];

// 修改字符串的最后一个字符
$str = 'Look at the sea';
$str[strlen($str)-1] = 'e';
?>
```

字符串下标必须为整数或可转换为整数的字符串，否则会发出警告（以前类似"foo" 的下标会转换成 0)。

```
<?php
$str = 'abc';

var_dump($str['1']);
var_dump(isset($str['1']));

var_dump($str['1.0']);
var_dump(isset($str['1.0']));

var_dump($str['x']);
var_dump(isset($str['x']));

var_dump($str['1x']);
var_dump(isset($str['1x']));
?>
```

以上例程在 PHP 5.3 中的输出：

```
string(1) "b"
bool(true)
string(1) "b"
bool(true)
string(1) "a"
bool(true)
string(1) "b"
bool(true)
```

以上例程在 PHP 5.4 中的输出：

```
string(1) "b"
bool(true)
```

```
Warning: Illegal string offset '1.0' in /tmp/t.php on line 7
string(1) "b"
bool(false)

Warning: Illegal string offset 'x' in /tmp/t.php on line 9
string(1) "a"
bool(false)
string(1) "b"
bool(false)
```

字符串可以用 '.'（点）运算符进行拼接，但是'+'（加号）运算符则没有这个功能。

- 正则表达式函数
- Perl 兼容正则表达式函数。
- URL 字符串函数
- 加密/解密字符串函数（mcrypt 和 mhash）。

17.5.7 String Casting

一个值可以通过在其前面加上 (string) 或用 strval() 函数来转变成字符串。

在一个需要字符串的表达式中，会自动转换为 string。例如，在使用函数 echo 或 print 时或在一个变量和一个 string 进行比较时就会发生这种转换。

一个布尔值 boolean 的 TRUE 被转换成 string 的"1"。Boolean 的 FALSE 被转换成""（空字符串）。这种转换可以在 boolean 和 string 之间相互进行。

一个整数 integer 或浮点数 float 被转换为数字的字面样式的 string（包括 float 中的指数部分）。使用指数计数法的浮点数（4.1E+6）也可转换。

数组 array 总是转换成字符串"Array"，因此 echo 和 print 无法显示出该数组的内容。要显示某个单元，可以用 `echo $arr['foo']` 这种结构。

- 在 PHP 4 中，对象 object 总是被转换成字符串"Object"。
- 为了得到对象的类的名称，可以用 get_class() 函数。
- 自 PHP 5 起，适当时可以用 `__toString` 方法。

资源 resource 总会被转变成"Resource id #1" 这种结构的字符串，其中的 1 是 PHP 在运行时分配给该 resource 的唯一值。注意，不要依赖此结构，可能会有变更。要得到一个 resource 的类型，可以用函数 get_resource_type()。

NULL 总是被转变成空字符串。

直接把 array，object 或 resource 转换成 string 不会得到除了其类型之外的任何有用信息。可以使用函数 print_r() 和 var_dump() 列出这些类型的内容。

大部分的 PHP 值可以转变成 string 来永久保存，这种操作被称作串行化（serialization），可以用函数 serialize() 来实现。如果 PHP 引擎设定支持 WDDX，PHP 值也可被串行化为格式良好的 XML 文本。

当一个字符串被当作一个数值来取值，其结果和类型如下：

- 如果该字符串没有包含 `'.'`,`'e'` 或 `'E'` 并且其数字值在整型的范围之内（由 PHP_INT_MAX 所定义），该字符串将被当成 integer 来取值。其它所有情况下都被作为 float 来取值。
- 该字符串的开始部分决定了它的值。如果该字符串以合法的数值开始，则使用该数值，否则其值为 0（零）。合法数值由可选的正负号，后面跟着一个或多个数字（可能有小数点），再跟着可选的指数部分。指数部分由 `'e'` 或 `'E'` 后面跟着一个或多个数字构成。

```
<?php
$foo = 1 + "10.5";           // $foo is float (11.5)
$foo = 1 + "-1.3e3";         // $foo is float (-1299)
$foo = 1 + "bob-1.3e3";      // $foo is integer (1)
$foo = 1 + "bob3";           // $foo is integer (1)
$foo = 1 + "10 Small Pigs";  // $foo is integer (11)
$foo = 4 + "10.2 Little Piggies"; // $foo is float (14.2)
$foo = "10.0 pigs " + 1;     // $foo is float (11)
$foo = "10.0 pigs " + 1.0;   // $foo is float (11)
?>
```

上述示例可以通过复制/粘贴到下面的代码中来显示：

```
<?php
echo "\$foo==$foo; type is " . gettype ($foo) . "<br />\n";
?>
```

PHP 使用函数 ord() 和 chr() 实现 ASCII 码和字符间的转换，而不是像在 C 语言中的那样，通过将一个字符转换成整数以得到其 ASCII 码。

17.5.8 String Structure

PHP 中的 string 的实现方式是一个由字节组成的数组再加上一个整数指明缓冲区长度，PHP 没有说明如何将字节转换成字符的信息，由程序员来决定。

字符串由什么值来组成并无限制。特别的，其值为 `0`（“NUL bytes”）的字节可以处于字符串任何位置，不过某些非“二进制安全”的函数也许会把 NUL 字节之后的数据全都忽略。

字符串类型的这种特性解释了为什么 PHP 中没有单独的“byte”类型——已经用字符串来代替了，返回非文本值的函数（例如从网络套接字读取的任意数据）仍会返回字符串。

PHP 并不特别指明字符串的编码，那字符串到底是怎样编码的呢？例如字符串 `"á"` 到底是等于 `"\xE1"`（ISO-8859-1），`"\xC3\xA1"`（UTF-8，C form），`"\x61\xCC\x81"`（UTF-8，D form）还是任何其它可能的表达呢？答案是字符串会被按照该脚本文件相同的编码方式来编码。

如果一个脚本的编码是 ISO-8859-1，则其中的字符串也会被编码为 ISO-8859-1，以此类推。不过这并不适用于激活了 Zend Multibyte 时，此时脚本可以是以任何方式编码的（明确指定或被自动检测）然后被转换为某种内部编码，然后字符串将被用此方式编码。

注意，PHP 脚本的编码有一些约束（如果激活了 Zend Multibyte 则是其内部编码），这意味着此编码应该是 ASCII 的兼容超集（例如 UTF-8 或 ISO-8859-1）。不过要注意，依赖状态的编码其中相同的字节值可以用于首字母和非首字母而转换状态，这可能会造成问题。

操作文本的函数必须假定字符串是如何编码的。不幸的是，PHP 关于此的函数有很多变种：

- 某些函数假定字符串是以单字节编码的，但并不需要将字节解释为特定的字符。例如 substr()，strpos()，strlen() 和 strcmp()。理解这些函数的另一种方法是它们作用于内存缓冲区，即按照字节和字节下标操作。
- 某些函数被传递入了字符串的编码方式，也可能会假定默认无此信息。例如 htmlentities() 和 mbstring 扩展中的大部分函数。
- 其它函数使用了当前区域（见 setlocale()），但是逐字节操作。例如 strcasecmp()，strtoupper() 和 ucfirst()。这意味着这些函数只能用于单字节编码，而且编码要与区域匹配。例如 `strtoupper("á")` 在区域设定正确并且 `á` 是单字节编码时会返回 `"Á"`。如果是用 UTF-8 编码则不会返回正确结果，其结果根据当前区域有可能返回损坏的值。
- 最后一些函数会假定字符串是使用某特定编码的（通常是 UTF-8）。intl 扩展和 PCRE（上例中仅在使用了 u 修饰符时）扩展中的大部分函数都是这样。尽管这是由于其特殊用途，utf8_decode() 会假定 UTF-8 编码而 utf8_encode() 会假定 ISO-8859-1 编码。

在编写能够正确使用 Unicode 的程序时，必须小心地避免那些可能会损坏数据的函数。例如，尽量使用来自于 intl 和 mbstring 扩展的函数。

不过，使用能处理 Unicode 编码的函数只是个开始，不管用何种语言提供的函数，最基本的还是了解 Unicode 规范。

17.6 Array

数组是 PHP 的基本数据类型，而且 PHP 提供了丰富的数组处理函数和方法（例如排序函数、替换函数、回调函数等），而且 PHP 数组函数还可以实现堆栈和队列等数据结构。

PHP 中的数组实际上是一个有序映射，映射本身是一种把 values 关联到 keys 的类型，但是 PHP 数组类型在很多方面做了优化，可以将其当作真正的数组，或列表（向量），散列表（是映射的一种实现），字典，集合，栈，队列等来使用。

数组元素的值也可以是另一个数组，因此树形结构和多维数组也是允许的。

- `array()`
  数组中任何都有固定的键名（索引）和值，用户可以用 array() 语言结构来新建一个数组，它接受任意数量用逗号分隔的 `键（key）=> 值（value）` 对。

```
array(  key =>  value
      , ...
      )
// 键（key）可是是一个整数 integer 或字符串 string
// 值（value）可以是任意类型的值
```

  最后一个数组单元之后的逗号可以省略，通常用于单行数组定义中，例如常用 `array(1, 2)` 而不是 `array(1, 2, )`。对多行数组定义通常可以保留最后一个逗号，这样要添加一个新单元时更方便。
- `[]` 用户也可以使用短数组定义语法，用 `[]` 替代 `array()`。

```
<?php
$array = array(
    "foo" => "bar",
    "bar" => "foo",
);

// 自 PHP 5.4 起
$array = [
    "foo" => "bar",
    "bar" => "foo",
];
?>
```

其中，key 可以是 integer 或者 string，value 可以是任意类型。

此外，key 会有如下的强制转换：

- 包含有合法整型值的字符串会被转换为整型。例如键名 `"8"` 实际会被储存为 `8`。但是 `"08"` 则不会强制转换，因为其不是一个合法的十进制数值。
- 浮点数也会被转换为整型，意味着其小数部分会被舍去。例如键名 `8.7` 实际会被储存为 `8`。
- 布尔值也会被转换成整型。即键名 `true` 实际会被储存为 `1` 而键名 `false` 会被储存为 `0`。
- Null 会被转换为空字符串，即键名 `null` 实际会被储存为 `""`。
- 数组和对象不能被用为键名，否则导致警告：`Illegal offset type`。

如果在数组定义中多个单元都使用了同一个键名，则只使用了最后一个，之前的都被覆盖了。

```php
<?php
$array = array(
    1    => "a",
    "1"  => "b",
    1.5  => "c",
    true => "d",
);
var_dump($array);
?>
```

以上例程会输出：

```
array(1) {
  [1]=>
  string(1) "d"
}
```

上例中所有的键名都被强制转换为 1，则每一个新单元都会覆盖前一个的值，最后剩下的只有一个 `"d"`。

PHP 数组内部使用类似 C 语言中“指针”的机制，可以便于对数组的管理和操作，索引、值和内部的指针是数组的重要组成部分。

PHP 数组不仅可以使用数字索引，也可以使用字符串键名的方式，这样就可以直接获取数组所有的键名和值，也可以方便地交换数组中的键名和值。

PHP 实际并不区分索引数组和关联数组，这样 PHP 数组才可以同时含有 integer 和 string 类型的键名。

- 如果对给出的值没有指定键名，则取当前最大的整数索引值，而新的键名将是该值加一。
- 如果指定的键名已经有了值，则该值会被覆盖。

```php
<?php
$array = array(
    "foo" => "bar",
    "bar" => "foo",
    100   => -100,
    -100 => 100,
);
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  ["foo"]=>
  string(3) "bar"
  ["bar"]=>
  string(3) "foo"
  [100]=>
  int(-100)
  [-100]=>
  int(100)
}
```

key 为可选项。如果未指定，PHP 将自动使用之前用过的最大 integer 键名加上 1 作为新的键名。

```
<?php
$array = array("foo", "bar", "hallo", "world");
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  [0]=>
  string(3) "foo"
  [1]=>
  string(3) "bar"
  [2]=>
  string(5) "hallo"
  [3]=>
  string(5) "world"
}
```

还可以只对某些单元指定键名而对其它的空置：

```
<?php
$array = array(
        "a",
        "b",
   6 => "c",
        "d",
```

```
);
var_dump($array);
?>
```

以上例程会输出：

```
array(4) {
  [0]=>
  string(1) "a"
  [1]=>
  string(1) "b"
  [6]=>
  string(1) "c"
  [7]=>
  string(1) "d"
}
```

可以看到最后一个值”d” 被自动赋予了键名 7，这是由于之前最大的整数键名是 6。

17.6.1 Array Operation

数组单元可以通过 `array[key]` 语法来访问。

```
<?php
$array = array(
    "foo" => "bar",
    42    => 24,
    "multi" => array(
         "dimensional" => array(
             "array" => "foo"
         )
    )
);

var_dump($array["foo"]);
var_dump($array[42]);
var_dump($array["multi"]["dimensional"]["array"]);
?>
```

以上例程会输出：

```
string(3) "bar"
```

```
int(24)
string(3) "foo"
```

方括号和花括号可以互换使用来访问数组单元（例如 $array[42] 和 $array{42} 在上例中效果相同）。

- 可以用数组间接引用函数或方法调用的结果。
- 可以用数组间接引用一个数组原型。

```
<?php
function getArray() {
    return array(1, 2, 3);
}

// on PHP 5.4
$secondElement = getArray()[1];

// previously
$tmp = getArray();
$secondElement = $tmp[1];

// or
list(, $secondElement) = getArray();
?>
```

试图访问一个未定义的数组键名与访问任何未定义变量一样，都会导致 E_NOTICE 级别错误信息，其结果为 NULL。

可以通过明确地设定其中的值来修改一个已有数组，这是通过在方括号内指定键名来给数组赋值实现的。也可以省略键名，在这种情况下给变量名加上一对空的方括号（`[]`）。

```
$arr[key] = value;
$arr[] = value;
// key 可以是 integer 或 string
// value 可以是任意类型的值
```

如果 `$arr` 还不存在，将会新建一个，这也是另一种新建数组的方法。不过并不鼓励这样做，因为如果 `$arr` 已经包含有值（例如来自请求变量的 string）则此值会保留而 `[]` 实际上代表着字符串访问运算符。初始化变量的最好方式是直接给其赋值。

- 要修改某个值，通过其键名给该单元赋一个新值。
- 要删除某键值对，对其调用 unset() 函数。

```
<?php
```

```
$arr = array(5 => 1, 12 => 2);

$arr[] = 56;  // This is the same as $arr[13] = 56;
              // at this point of the script

$arr["x"] = 42; // This adds a new element to
              // the array with key "x"

unset($arr[5]); // This removes the element from the array

unset($arr);  // This deletes the whole array
?>
```

如上所述，如果给出方括号但没有指定键名，则取当前最大整数索引值，新的键名将是该值加上 1（但是最小为 0）。如果当前还没有整数索引，则键名将为 0。

注意，这里所使用的最大整数键名不一定当前就在数组中，它只要在上次数组重新生成索引后曾经存在过就行了。

```
<?php
// 创建一个简单的数组
$array = array(1, 2, 3, 4, 5);
print_r($array);

// 现在删除其中的所有元素，但保持数组本身不变：
foreach ($array as $i => $value) {
    unset($array[$i]);
}
print_r($array);

// 添加一个单元（注意新的键名是 5，而不是你可能以为的 0）
$array[] = 6;
print_r($array);

// 重新索引：
$array = array_values($array);
$array[] = 7;
print_r($array);
?>
```

以上例程会输出：

```
Array
```

```
(
    [0] => 1
    [1] => 2
    [2] => 3
    [3] => 4
    [4] => 5
)
Array
(
)
Array
(
    [5] => 6
)
Array
(
    [0] => 6
    [1] => 7
)
```

17.6.2 Array Function

- unset() 函数允许删除数组中的某个键，但是注意数组将不会重建索引。
- 如果需要删除后重建索引，可以用 array_values() 函数。

```php
<?php
$a = array(1 => 'one', 2 => 'two', 3 => 'three');
unset($a[2]);
/* will produce an array that would have been defined as
   $a = array(1 => 'one', 3 => 'three');
   and NOT
   $a = array(1 => 'one', 2 =>'three');
*/

$b = array_values($a);
// Now $b is array(0 => 'one', 1 =>'three')
?>
```

foreach 控制结构是专门用于数组的，而且它提供了一个简单的方法来遍历数组。

```
<?php
$colors = array('red', 'blue', 'green', 'yellow');

foreach ($colors as $color) {
    echo "Do you like $color?\n";
}

?>
```

以上例程会输出：

```
Do you like red?
Do you like blue?
Do you like green?
Do you like yellow?
```

应该始终在用字符串表示的数组索引上加上引号，例如用 $foo['bar']$□□□foo[bar]。例如，可能在老的脚本中见过如下语法：

```
<?php
$foo[bar] = 'enemy';
echo $foo[bar];
// etc
?>
```

这样是错的，但可以正常运行。那么为什么错了呢？原因是此代码中有一个未定义的常量（bar）而不是字符串（'bar' －注意引号），而 PHP 可能会在以后定义此常量，不幸的是你的代码中有同样的名字。它能运行，是因为 PHP 自动将裸字符串（没有引号的字符串且不对应于任何已知符号）转换成一个其值为该裸字符串的正常字符串。例如，如果没有常量定义为 bar，PHP 将把它替代为'bar' 并使用之。

这并不意味着总是给键名加上引号。用不着给键名为常量或变量的加上引号，否则会使 PHP 不能解析它们。

```
<?php
error_reporting(E_ALL);
ini_set('display_errors', true);
ini_set('html_errors', false);
// Simple array:
$array = array(1, 2);
$count = count($array);
for ($i = 0; $i < $count; $i++) {
    echo "\nChecking $i: \n";
```

```
    echo "Bad: " . $array['$i'] . "\n";
    echo "Good: " . $array[$i] . "\n";
    echo "Bad: {$array['$i']}\n";
    echo "Good: {$array[$i]}\n";
}
?>
```

以上例程会输出:

```
Checking 0:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 1
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 1

Checking 1:
Notice: Undefined index:  $i in /path/to/script.html on line 9
Bad:
Good: 2
Notice: Undefined index:  $i in /path/to/script.html on line 11
Bad:
Good: 2
```

演示此行为的更多例子:

```
<?php
// Show all errors
error_reporting(E_ALL);

$arr = array('fruit' => 'apple', 'veggie' => 'carrot');

// Correct
print $arr['fruit']; // apple
print $arr['veggie']; // carrot

// Incorrect. This works but also throws a PHP error of level E_NOTICE because
// of an undefined constant named fruit
//
// Notice: Use of undefined constant fruit - assumed 'fruit' in...
```

```
print $arr[fruit]; // apple

// This defines a constant to demonstrate what's going on. The value 'veggie'
// is assigned to a constant named fruit.
define('fruit', 'veggie');

// Notice the difference now
print $arr['fruit']; // apple
print $arr[fruit]; // carrot

// The following is okay, as it's inside a string. Constants are not looked for
// within strings, so no E_NOTICE occurs here
print "Hello $arr[fruit]"; // Hello apple

// With one exception: braces surrounding arrays within strings allows constants
// to be interpreted
print "Hello {$arr[fruit]}"; // Hello carrot
print "Hello {$arr['fruit']}"; // Hello apple

// This will not work, and will result in a parse error, such as:
// Parse error: parse error, expecting T_STRING' or T_VARIABLE' or T_NUM_STRING'
// This of course applies to using superglobals in strings as well
print "Hello $arr['fruit']";
print "Hello $_GET['foo']";

// Concatenation is another option
print "Hello " . $arr['fruit']; // Hello apple
?>
```

当打开 error_reporting 来显示 E_NOTICE 级别的错误（将其设为 E_ALL）时将看到这些错误，默认情况下 error_reporting 被关闭不显示这些。

在方括号（“[”和“]”）之间必须有一个表达式，这意味着可以这样写：

```
<?php
echo $arr[somefunc($bar)];
?>
```

这是一个用函数返回值作为数组索引的例子。

PHP 也可以用已知常量。例如：

```
<?php
$error_descriptions[E_ERROR] = "A fatal error has occured";
$error_descriptions[E_WARNING] = "PHP issued a warning";
```

```
$error_descriptions[E_NOTICE] = "This is just an informal notice";
?>
```

注意，E_ERROR 也是个合法的标识符，就和第一个例子中的 bar 一样，但是上一个例子实际上和如下写法是一样的：

```
<?php
$error_descriptions[1] = "A fatal error has occured";
$error_descriptions[2] = "PHP issued a warning";
$error_descriptions[8] = "This is just an informal notice";
?>
```

因为 E_ERROR 等于 1，不过以后想新增一个常量或者关键字，或者用户可能希望以后在自己的程序中引入新的常量时就会产生麻烦，现在已经不能这样用 empty 和 default 这两个词了，因为它们是保留字。

在双引号字符串中，不给索引加上引号是合法的，因此”$foo[bar]” 是合法的（在实际测试中，这么做确实可以访问数组的该元素，但是会报一个常量未定义的 notice）。

强烈建议不要使用 $foo[bar] 这样的写法，而要使用 $foo['bar'] 来访问数组中元素。

17.6.3 Array Casting

对于任意 integer，float，string，boolean 和 resource 类型，如果将一个值转换为数组，将得到一个仅有一个元素的数组，其下标为 0，该元素即为此标量的值。换句话说，(array)$scalarValue 与 array($scalarValue) 完全一样。

如果一个 object 类型转换为 array，则结果为一个数组，其单元为该对象的属性。键名将为成员变量名，不过有几点例外：整数属性不可访问；私有变量前会加上类名作前缀；保护变量前会加上一个’*’ 做前缀。这些前缀的前后都各有一个 NULL 字符，这会导致一些不可预知的行为：

```
<?php

class A {
   private $A; // This will become '\0A\0A'
}

class B extends A {
   private $A; // This will become '\0B\0A'
   public $AA; // This will become 'AA'
}

var_dump((array) new B());
?>
```

上例会有两个键名为'AA'，不过其中一个实际上是'\0A\0A'。
将 NULL 转换为 array 会得到一个空的数组。

17.6.4 Array Comparison

可以用 array_diff() 和数组运算符来比较数组。

```
<?php
// This:
$a = array( 'color' => 'red',
          'taste' => 'sweet',
          'shape' => 'round',
          'name' => 'apple',
          4       // key will be 0
        );

$b = array('a', 'b', 'c');

// . . .is completely equivalent with this:
$a = array();
$a['color'] = 'red';
$a['taste'] = 'sweet';
$a['shape'] = 'round';
$a['name'] = 'apple';
$a[]       = 4;       // key will be 0

$b = array();
$b[] = 'a';
$b[] = 'b';
$b[] = 'c';

// After the above code is executed, $a will be the array
// array('color' => 'red', 'taste' => 'sweet', 'shape' => 'round',
// 'name' => 'apple', 0 => 4), and $b will be the array
// array(0 => 'a', 1 => 'b', 2 => 'c'), or simply array('a', 'b', 'c').
?>
```

使用 array() 的示例如下：

```
<?php
// Array as (property-)map
$map = array( 'version' => 4,
           'OS'       => 'Linux',
```

```
            'lang'     => 'english',
            'short_tags' => true
          );

// strictly numerical keys
$array = array( 7,
              8,
              0,
              156,
              -10
            );
// this is the same as array(0 => 7, 1 => 8, ...)

$switching = array(     10, // key = 0
                 5    => 6,
                 3    => 7,
                 'a'  => 4,
                         11, // key = 6 (maximum of integer-indices was 5)
                 '8'  => 2, // key = 8 (integer!)
                 '02' => 77, // key = '02'
                 0    => 12 // the value 10 will be overwritten by 12
               );

// empty array
$empty = array();
?>
```

直接改变数组的值自 PHP 5 起可以通过引用传递来做到。之前的版本需要需要采取变通的方法：

```
<?php
// PHP 5
foreach ($colors as &$color) {
    $color = strtoupper($color);
}
unset($color); /* ensure that following writes to
$color will not modify the last array element */

// Workaround for older versions
foreach ($colors as $key => $color) {
    $colors[$key] = strtoupper($color);
}
```

```
print_r($colors);
?>
```

以上例程会输出：

```
Array
(
    [0] => RED
    [1] => BLUE
    [2] => GREEN
    [3] => YELLOW
)
```

本例生成一个下标从 1 开始的数组。

```
<?php
$firstquarter = array(1 => 'January', 'February', 'March');
print_r($firstquarter);
?>
```

以上例程会输出：

```
Array
(
    [1] => 'January'
    [2] => 'February'
    [3] => 'March'
)
```

```
<?php
// fill an array with all items from a directory
$handle = opendir('.');
while (false !== ($file = readdir($handle))) {
   $files[] = $file;
}
closedir($handle);
?>
```

数组是有序的，也可以使用不同的排序函数来改变顺序，还可以用 count() 函数来数出数组中元素的个数。

```php
<?php
sort($files);
print_r($files);
?>
```

17.6.5 Multidimensional Array

因为数组中的值可以为任意值，也可以是另一个数组，这样可以产生递归或多维数组。

```php
<?php
$fruits = array ( "fruits" => array ( "a" => "orange",
                                      "b" => "banana",
                                      "c" => "apple"
                                    ),
                  "numbers" => array ( 1,
                                       2,
                                       3,
                                       4,
                                       5,
                                       6
                                     ),
                  "holes"  => array (    "first",
                                       5 => "second",
                                            "third"
                                     )
                );

// Some examples to address values in the array above
echo $fruits["holes"][5]; // prints "second"
echo $fruits["fruits"]["a"]; // prints "orange"
unset($fruits["holes"][0]); // remove "first"

// Create a new multi-dimensional array
$juices["apple"]["green"] = "good";
?>
```

数组 (Array) 的赋值总是会涉及到值的拷贝，使用引用运算符通过引用来拷贝数组。

```php
<?php
$arr1 = array(2, 3);
$arr2 = $arr1;
$arr2[] = 4; // $arr2 is changed,
           // $arr1 is still array(2, 3)
```

```
$arr3 = &$arr1;
$arr3[] = 4; // now $arr1 and $arr3 are the same
?>
```

17.7 Object

17.7.1 Object Initialization

要创建一个新的对象 object，使用 new 语句实例化一个类：

```
<?php
class foo
{
   function do_foo()
   {
      echo "Doing foo.";
   }
}

$bar = new foo;
$bar->do_foo();
?>
```

17.7.2 Object Casting

- 如果将一个对象转换成对象，它将不会有任何变化。
- 如果其它任何类型的值被转换成对象，将会创建一个内置类 stdClass 的实例。
- 如果该值为 NULL，则新的实例为空。
- 数组转换成对象将使键名成为属性名并具有相对应的值。

对于任何其它的值，名为 scalar 的成员变量将包含该值。

```
?php
$obj = (object) 'ciao';
echo $obj->scalar; // outputs 'ciao'
?>
```

17.8 Resource

资源是通过专门的函数来建立和使用的。

资源 resource 是一种特殊变量，保存了到外部资源的一个引用。

PHP 4 Zend 引擎引进了引用计数系统，可以自动检测到一个资源不再被引用了（和 Java 一样），这种情况下此资源使用的所有外部资源都会被垃圾回收系统释放，因此很少需要手工释放内存。

只有持久数据库连接比较特殊，它们不会被垃圾回收系统销毁。

17.8.1 Resource Casting

由于资源类型变量保存有为打开文件、数据库连接、图形画布区域等的特殊句柄，因此将其它类型的值转换为资源没有意义。

17.9 NULL

特殊的 NULL 值表示一个变量没有值，NULL 类型唯一可能的值就是 NULL。

在下列情况下一个变量被认为是 NULL:

- 被赋值为 NULL。
- 尚未被赋值。
- 被 unset()。

NULL 类型只有一个值，就是不区分大小写的常量 NULL。

```
<?php
$var = NULL;
?>
```

17.9.1 NULL Casting

使用 `(unset) $var` 将一个变量转换为 null 将不会删除该变量或 unset 其值，仅是返回 NULL 值而已。

17.10 Callback

callable 类型可以用来指定回调类型 callback。

`call_user_func()` 或 `usort()` 等函数可以接受用户自定义的回调函数作为参数，而且回调函数不止可以是简单函数，还可以是对象的方法（包括静态类方法）。

17.10.1 Callback Passing

一个 PHP 的函数以 string 类型传递其名称，可以使用任何内置或用户自定义函数（语言结构除外，例如 `array()`，`echo`，`empty()`，`eval()`，`exit()`，`isset()`，`list()`，`print` 或 `unset()`）。

一个已实例化的对象的方法被作为数组传递，下标 0 包含该对象，下标 1 包含方法名。

静态类方法也可以不经实例化该类的对象就进行传递，只要在下标 0 中包含类名而不是对象。自 PHP 5.2.3 起，也可以传递 `'ClassName::methodName'`。

除了普通的用户自定义函数外，`create_function()` 可以用来创建一个匿名回调函数，也可以传递 closure 给回调参数。

回调函数的示例如下：

```
<?php

// An example callback function
function my_callback_function() {
    echo 'hello world!';
}

// An example callback method
class MyClass {
    static function myCallbackMethod() {
        echo 'Hello World!';
    }
}

// Type 1: Simple callback
call_user_func('my_callback_function');

// Type 2: Static class method call
call_user_func(array('MyClass', 'myCallbackMethod'));

// Type 3: Object method call
$obj = new MyClass();
call_user_func(array($obj, 'myCallbackMethod'));

// Type 4: Static class method call (As of PHP 5.2.3)
call_user_func('MyClass::myCallbackMethod');

// Type 5: Relative static class method call (As of PHP 5.3.0)
class A {
```

```
    public static function who() {
        echo "A\n";
    }
}

class B extends A {
    public static function who() {
        echo "B\n";
    }
}

call_user_func(array('B', 'parent::who')); // A
?>
```

使用 Closure 的示例如下：

```
<?php
// Our closure
$double = function($a) {
    return $a * 2;
};

// This is our range of numbers
$numbers = range(1, 5);

// Use the closure as a callback here to
// double the size of each element in our
// range
$new_numbers = array_map($double, $numbers);

print implode(' ', $new_numbers);
?>
```

以上例程会输出：

```
2 4 6 8 10
```

在 PHP 4 中，需要使用一个引用来创建一个指向具体对象的回调函数，而不是一个拷贝。

在函数中注册有多个回调内容时 (如使用 call_user_func() 与 call_user_func_array())，如在前一个回调中有未捕获的异常，其后的将不再被调用。

17.11 Pseudo-types

- mixed
  mixed 说明一个参数可以接受多种不同的（但不一定是所有的）类型。例如，gettype() 可以接受所有的 PHP 类型，str_replace() 可以接受字符串和数组。
- number
  number 说明一个参数可以是 integer 或者 float。
- callback
  callable 类型和之前使用的 callback 伪类型，二者含义完全相同。
- void
  - void 作为返回类型意味着函数的返回值是无用的。
  - void 作为参数列表意味着函数不接受任何参数。
- ...
  在函数原型中，$... 表示等等的意思。当一个函数可以接受任意个参数时使用此变量名。

17.12 Type Juggling

PHP 在变量定义中不需要（或不支持）明确的类型定义，变量类型是根据使用该变量的上下文所决定的。也就是说，如果把一个字符串值赋给变量 $var，$var 就成了一个字符串。如果又把一个整型值赋给 $var，那它就成了一个整数。

PHP 的自动类型转换的一个例子是加法运算符“+”。如果任何一个操作数是浮点数，则所有的操作数都被当成浮点数，结果也是浮点数。否则操作数会被解释为整数，结果也是整数。注意，这里并没有改变这些操作数本身的类型，改变的仅是这些操作数如何被求值以及表达式本身的类型。

```
<?php
$foo = "0"; // $foo 是字符串 (ASCII 48)
$foo += 2; // $foo 现在是一个整数 (2)
$foo = $foo + 1.3; // $foo 现在是一个浮点数 (3.3)
$foo = 5 + "10 Little Piggies"; // $foo 是整数 (15)
$foo = 5 + "10 Small Pigs"; // $foo 是整数 (15)
?>
```

自动转换为数组的行为目前没有定义。

此外，由于 PHP 支持使用和数组下标同样的语法访问字符串下标，因此以下例子在所有 PHP 版本中都有效：

```
<?php
```

```
$a   = 'car'; // $a is a string
$a[0] = 'b'; // $a is still a string
echo $a;     // bar
?>
```

PHP 会判断变量类型并在需要时进行转换（通常情况下），因此在某一时刻给定的变量是何种类型并不明显。PHP 包括几个函数可以判断变量的类型，例如 gettype()，is_array()，is_float()，is_int()，is_object() 和 is_string()。

17.13 Type Casting

在实际应用中，经常使用不同类型的变量来满足不同应用程序接口的需求，因此需要对变量进行类型识别或转换。

- gettype() 可以查看一个变量的数据类型。
- var_dump() 可以查看一个变量的数据类型和值。
- print_r() 可以查看一个变量的值，并且输出更直观的数据结果。
- var_export() 可以返回一个变量的字符串表示。

PHP 中的类型强制转换和 C 中的非常相似，在要转换的变量之前加上用括号括起来的目标类型。或者，使用 settype() 函数来将目标数据类型作为参数，并输出转换后的变量。

```
<?php
$foo = 10; // $foo is an integer
$bar = (boolean) $foo; // $bar is a boolean
?>
```

PHP 允许的强制转换包括：

- (int), (integer) - 转换为整形 integer
- (bool), (boolean) - 转换为布尔类型 boolean
- (float), (double), (real) - 转换为浮点型 float
- (string) - 转换为字符串 string
- (array) - 转换为数组 array
- (object) - 转换为对象 object
- (unset) - 转换为 NULL

虽然浮点数和整型数可以相互转换，但是可能有精度损失。

- 整型转换为浮点型时没有精度损失；
- 浮点型转换为整型时自动舍弃小数部分。

如果一个浮点数超过整型数的有效范围，那么溢出的部分将会丢失，因此结果将是不确定的。

注意，在括号内允许有空格和制表符，所以下面两个例子功能相同：

```
<?php
$foo = (int) $bar;
$foo = ( int ) $bar;
?>
```

PHP 支持 (binary) 转换和 b 前缀转换。例如，将字符串文字和变量转换为二进制字符串：

```
<?php
$binary = (binary)$string;
$binary = b"binary string";
?>
```

可以将变量放置在双引号中的方式来代替将变量转换成字符串：

```
<?php
$foo = 10;          // $foo 是一个整数
$str = "$foo";      // $str 是一个字符串
$fst = (string) $foo; // $fst 也是一个字符串

// 输出 "they are the same"
if ($fst === $str) {
    echo "they are the same";
}
?>
```

有时在类型之间强制转换时确切地会发生什么可能不是很明显。

除此之外，PHP 还提供了更为具体的转换函数 intval()、floatval() 和 strval() 分别用来转换整型、浮点型和字符串类型。

为了获得一个指定精度的浮点数，可以参考 C 语言使用 sprintf()，并返回一个格式化了的字符串。

```
<?php
$foo = sprintf("%0.2f",3.1415926);
$foo = (float)sprintf("%0.2f",3.1415926);
?>
```

字符串在转换为数字时得到的数字是从字符串开始部分的数值型字符串，而且数值型字符串包括使用科学记数法表示的数字，因此 PHP 中的数值型字符串可以和数字变量进行混合运算。

数组和对象转换为字符串时的结果分别是“Array”和“Object”，因此试图以 echo 或 print 输出一个数组或字符串并不能得到预期的结果，应该使用 print_r() 或 var_dump() 进行输出。

如果将资源型数据转换为字符串，那么将得到类似于“Resource id#1”格式的字符串，其中 1 是 PHP 中资源的唯一标识符，可以使用 get_resource_type() 获得资源的类型。

```
<?php
$fp = fopen("foo", "w");
echo get_resource_type($fp);
?>
```

在将布尔值、数字或字符串转换为数组时，将得到一个以上述类型数据为元素的数组。

- 将一个 NULL 转换为数组将得到一个空数组；
- 将一个对象转换为数组时将使数组元素变为对象的属性，其键名为成员变量名。

Chapter 18

PHP Variable

PHP 中的所有变量都是以 $ 符号开始的，变量名是区分大小写的，而且与 PHP 中其它的标签一样遵循相同的规则。

一个有效的变量名由字母或者下划线开头，后面跟上任意数量的字母[1]，数字，或者下划线。

按照正常的正则表达式，PHP 变量将被表述为：`'[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*'`。

通常情况下，PHP 不会改变传递给脚本中的变量名，不过点（句号）不是 PHP 变量名中的合法字符，因此 PHP 将会自动将变量名中的点替换成下划线。

```
<?php
$varname.ext; /* 非法变量名 */
?>
```

PHP 解析器看到是一个名为 $varname 的变量，后面跟着一个字符串连接运算符，后面跟着一个裸字符串（即没有加引号的字符串，且不匹配任何已知的健名或保留字）`'ext'`，很明显这不是想要的结果。

变量用于存储值[2]，比如数字、字符串或函数的结果，这样我们就可以在脚本中多次使用它们了。

18.1 Type

与 C++ 等不同的是，不需要在 PHP 变量使用前确定该变量的类型，由上下文中变量所赋的值决定变量的类型。

```
<?php
```

[1]在此所说的字母是 a-z，A-Z，以及 ASCII 字符从 127 到 255（0x7f-0xff）。

[2]注：$this 是一个特殊的变量，它不能被赋值。

```
$var = 'Bob';
$Var = 'Joe';
echo "$var, $Var";   // 输出 "Bob, Joe"

$4site = 'not yet'; // 非法变量名；以数字开头
$_4site = 'not yet'; // 合法变量名；以下划线开头
$i站点is = 'mansikka'; // 合法变量名；可以用中文
?>
```

在决定了一个变量的类型后，不要轻易改动，以免发生错误。

下面是在 PHP 中设置变量的语法：

```
$var_name = value;
```

如果忘记在变量的前面的 $ 符号的话，变量将是无效的。

下面将创建一个存有字符串的变量，和一个存有数值的变量：

```
<?php
$txt = "Hello World!";
$number = 16;
?>
```

PHP 是一门松散类型的语言（Loosely Typed Language），不需要在设置变量之前声明该变量，也不必向 PHP 声明该变量的数据类型，根据变量被设置的方式，PHP 会自动地把变量转换为正确的数据类型。

在强类型的编程语言中，必须在使用前声明变量的类型和名称，而在 PHP 中，变量会在使用时被自动声明。

PHP 中变量的命名规则如下：

- 变量名必须以字母或下划线”_” 开头。
- 变量名只能包含字母数字字符以及下划线。
- 变量名不能包含空格。如果变量名由多个单词组成，那么应该使用下划线进行分隔（比如 `$my_string`），或者以大写字母开头（比如 `$myString`）。

18.2 Value

变量默认总是传值赋值。那也就是说，当将一个表达式的值赋予一个变量时，整个原始表达式的值被赋值到目标变量。这意味着，当一个变量的值赋予另外一个变量时，改变其中一个变量的值，将不会影响到另外一个变量。

18.3 Reference

PHP 也提供了另外一种方式给变量赋值：引用赋值。

- 新的变量简单的引用（换言之，“成为其别名”或者“指向”）了原始变量。
- 改动新的变量将影响到原始变量，反之亦然。

使用引用赋值，简单地将一个 & 符号加到将要赋值的变量前（源变量）。例如，下列代码片断将输出“My name is Bob”两次：

```
<?php
$foo = 'Bob';           // 将 'Bob' 赋给 $foo
$bar = &$foo;           // 通过 $bar 引用 $foo
$bar = "My name is $bar"; // 修改 $bar 变量
echo $bar;
echo $foo;              // $foo 的值也被修改
?>
```

必须指出的是，只有有名字的变量才可以引用赋值。

```
<?php
$foo = 25;
$bar = &$foo;    // 合法的赋值
$bar = &(24 * 7); // 非法；引用没有名字的表达式

function test()
{
   return 25;
}

$bar = &test(); // 非法
?>
```

虽然在 PHP 中并不需要初始化变量，不过对变量进行初始化仍然是个好习惯，而且未初始化的变量只有其类型的默认值。

- 布尔类型的变量默认值是 FALSE；
- 整形和浮点型变量默认值是零；
- 字符串型变量（例如用于 echo 中）默认值是空字符串；
- 数组变量的默认值是空数组。

未初始化变量的默认值如下：

```
<?php
// Unset AND unreferenced (no use context) variable; outputs NULL
var_dump($unset_var);
```

```
// Boolean usage; outputs 'false' (See ternary operators for more on this syntax)
echo($unset_bool ? "true\n" : "false\n");

// String usage; outputs 'string(3) "abc"'
$unset_str .= 'abc';
var_dump($unset_str);

// Integer usage; outputs 'int(25)'
$unset_int += 25; // 0 + 25 => 25
var_dump($unset_int);

// Float/double usage; outputs 'float(1.25)'
$unset_float += 1.25;
var_dump($unset_float);

// Array usage; outputs array(1) { [3]=> string(3) "def" }
$unset_arr[3] = "def"; // array() + array(3 => "def") => array(3 => "def")
var_dump($unset_arr);

// Object usage; creates new stdClass object (see http://www.php.net/manual/en/reserved.
    classes.php)
// Outputs: object(stdClass)#1 (1) { ["foo"]=> string(3) "bar" }
$unset_obj->foo = 'bar';
var_dump($unset_obj);
?>
```

依赖未初始化变量的默认值在某些情况下会有问题，例如把一个文件包含到另一个之中时碰上相同的变量名。

另外，把 register_globals 打开是一个主要的安全隐患。使用未初始化的变量会发出 E_NOTICE 错误，但是在向一个未初始化的数组附加单元时不会。

isset() 语言结构可以用来检测一个变量是否已被初始化。

18.4 Variable scope

变量的范围即它定义的上下文背景（也就是它的生效范围），因此变量的范围也称为变量的作用域。

大部分的 PHP 变量只有一个单独的范围，而且这个单独的范围跨度同样包含了 include 和 require 引入的文件。

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

```
include 'b.inc';
?>
```

这里变量 $a 将会在包含文件 b.inc 中生效。

在用户自定义函数中，一个局部函数范围将被引入，任何用于函数内部的变量按默认情况都将被限制在局部函数范围内。

```
<?php
$a = 1; /* global scope */

function Test()
{
    echo $a; /* reference to local scope variable */
}

Test();
?>
```

这个脚本不会有任何输出，因为 echo 语句引用了一个局部版本的变量 $a，而且在这个范围内，它并没有被赋值。

- 在函数内部定义的局部变量（或参数，引用除外），不能访问函数的外部；
- 在函数外部定义的全局变量，不能访问函数的内部。

18.4.1 Global Variable

C 语言的全局变量在函数中自动生效，除非被局部变量覆盖，因此在 PHP 中使用全局变量可能引起某些问题。

在 PHP 函数中使用全局变量时必须声明为 global。

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    global $a, $b;

    $b = $a + $b;
}

Sum();
echo $b;
?>
```

以上脚本的输出将是“3”。在函数中声明了全局变量 $a 和 $b 之后，对任一变量的所有引用都会指向其全局版本。

- 在函数中声明了全局变量后，对任一变量的所有引用都会指向其全局版本。
- 对于一个函数能够声明的全局变量的最大个数，PHP 没有限制。

PHP 无法在函数内部自动地使用全局变量，必须在函数内部使用 global 关键字将变量声明为全局的，才能在函数内部使用全局变量。

在全局范围内访问变量的第二个办法是使用 PHP 自定义的特殊的 $GLOBALS 数组。

$GLOBALS 数组是一个在 PHP 程序运行时自动创建的特殊变量，而且 $GLOBALS 数组中的元素和外部变量是一一对应的。

```
<?php
$a = 1;
$b = 2;

function Sum()
{
    $GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}

Sum();
echo $b;
?>
```

$GLOBALS 是一个关联数组，每一个变量为一个元素，键名对应变量名，值对应变量的内容。

$GLOBALS 之所以在全局范围内存在，是因为 $GLOBALS 是一个超全局变量。以下范例显示了超全局变量的用处：

```
<?php
function test_global()
{
    // 大多数的预定义变量并不 "super"，它们需要用 'global' 关键字来
    // 使它们在函数的本地区域中有效。
    global $HTTP_POST_VARS;

    echo $HTTP_POST_VARS['name'];

    // Superglobals 在任何范围内都有效，它们并不需要 'global' 声明。
    echo $_POST['name'];
}
?>
```

18.4.2 Static Variable

默认情况下，函数执行结束后，其内部的局部变量自动从内存空间消失，可以使用静态变量将函数的内部变量持久地保留在内存中。

静态变量（static variable）仅在局部函数域中存在，当程序执行离开此作用域时，其值并不丢失。

在程序再次执行时，静态变量将接续上一次的结果继续运算，因此在某些应用场景（例如递归运算）中使用静态变量是很有用的。

```
<?php
function Test()
{
    $a = 0;
    echo $a;
    $a++;
}
?>
```

本函数没什么用处，因为每次调用时都会将 $a 的值设为 0 并输出 0。将变量加一的 $a++ 没有作用，因为一旦退出本函数则变量 $a 就不存在了。

如果要写一个不会丢失本次计数值的计数函数，要将变量 $a 定义为静态的：

```
<?php
function test()
{
    static $a = 0;
    echo $a;
    $a++;
}
?>
```

现在，变量 $a 仅在第一次调用 test() 函数时被初始化，之后每次调用 test() 函数都会输出 $a 的值并加一。

静态变量同时提供了一种处理递归函数的方法，递归函数是一种调用自己的函数，必须确保有充分的方法来中止递归否则很可能会无穷递归下去。以下这个简单的函数递归计数到 10，使用静态变量 $count 来判断何时停止：

```
<?php
function test()
{
    static $count = 0;

    $count++;
```

```
    echo $count;
    if ($count < 10) {
        test();
    }
    $count--;
}
?>
```

静态变量可以按照上面的例子声明，如果在声明中用表达式的结果对其赋值会导致解析错误。

```
<?php
function foo(){
    static $int = 0;       // correct
    static $int = 1+2;     // wrong (as it is an expression)
    static $int = sqrt(121); // wrong (as it is an expression too)

    $int++;
    echo $int;
}
?>
```

静态声明是在编译时解析的，在函数之外使用 global 关键字不算错，可以用于在一个函数之内包含文件时。

在 Zend 引擎 1 代（它驱动了 PHP4）中，对于变量的 static 和 global 定义是以引用的方式实现的。例如，在一个函数域内部用 global 语句导入的一个真正的全局变量实际上是建立了一个到全局变量的引用，不过有可能导致预料之外的行为。

```
<?php
function test_global_ref() {
    global $obj;
    $obj = &new stdclass;
}

function test_global_noref() {
    global $obj;
    $obj = new stdclass;
}

test_global_ref();
var_dump($obj);
test_global_noref();
var_dump($obj);
```

```
?>
```

以上例程会输出：

```
NULL
object(stdClass)(0) {
}
```

类似的行为也适用于 static 语句，引用并不是静态地存储的。

```
<?php
function &get_instance_ref() {
    static $obj;

    echo 'Static object: ';
    var_dump($obj);
    if (!isset($obj)) {
        // 将一个引用赋值给静态变量
        $obj = &new stdclass;
    }
    $obj->property++;
    return $obj;
}

function &get_instance_noref() {
    static $obj;

    echo 'Static object: ';
    var_dump($obj);
    if (!isset($obj)) {
        // 将一个对象赋值给静态变量
        $obj = new stdclass;
    }
    $obj->property++;
    return $obj;
}

$obj1 = get_instance_ref();
$still_obj1 = get_instance_ref();
echo "\n";
$obj2 = get_instance_noref();
$still_obj2 = get_instance_noref();
?>
```

以上例程会输出：

```
Static object: NULL
Static object: NULL

Static object: NULL
Static object: object(stdClass)(1) {
["property"]=>
int(1)
}
```

上例演示了当把一个引用赋值给一个静态变量时，第二次调用 `&get_instance_ref()` 函数时其值并没有被记住。

18.5 Variable Casting

通常情况下，PHP 可以自动判断变量类型并在需要时进行转换，因此在某一时刻给定的变量的类型并不明显。

PHP 提供了相关的函数来判断变量的当前类型。

- gettype()
- is_array()
- is_float()
- is_int()
- is_object()
- is_string()

18.6 Variable variables

可变变量名就是指一个变量的变量名可以动态的设置和使用。

一个普通的变量通过声明来设置，例如：

```
<?php
$a = 'hello';
?>
```

一个可变变量获取了一个普通变量的值作为这个可变变量的变量名。在上面的例子中 hello 使用了两个美元符号（$）以后，就可以作为一个可变变量的变量了。例如：

```
<?php
$$a = 'world';
?>
```

这时，两个变量都被定义了：$a 的内容是“hello”并且 $hello 的内容是“world”。

```
<?php
echo "$a ${$a}";
?>
```

与以下语句输出完全相同的结果：

```
<?php
echo "$a $hello";
?>
```

它们都会输出：`hello world`。

要将可变变量用于数组，必须解决一个模棱两可的问题，那就是当写下 $$a[1] 时，解析器需要知道是想要 $a[1] 作为一个变量呢，还是想要 $$a 作为一个变量并取出该变量中索引为 [1] 的值。解决此问题的语法是，对第一种情况用 ${$a[1]}，对第二种情况用 ${$a}[1]。

类的属性也可以通过可变属性名来访问。可变属性名将在该调用所处的范围内被解析。例如，对于 $foo->$bar 表达式，则会在本地范围来解析 $bar 并且其值将被用于 $foo 的属性名。对于 $bar 是数组单元时也是一样。

也可使用花括号来给属性名清晰定界。最有用是在属性位于数组中，或者属性名包含有多个部分或者属性名包含有非法字符时（例如来自 json_decode() 或 SimpleXML）。

```
<?php
class foo {
    var $bar = 'I am bar.';
    var $arr = array('I am A.', 'I am B.', 'I am C.');
    var $r   = 'I am r.';
}

$foo = new foo();
$bar = 'bar';
$baz = array('foo', 'bar', 'baz', 'quux');
echo $foo->$bar . "\n";
echo $foo->$baz[1] . "\n";

$start = 'b';
$end   = 'ar';
echo $foo->{$start . $end} . "\n";
```

```
$arr = 'arr';
echo $foo->$arr[1] . "\n";
echo $foo->{$arr}[1] . "\n";

?>
```

以上例程会输出：

```
I am bar.
I am bar.
I am bar.
I am r.
I am B.
```

注意，在 PHP 的函数和类的方法中，超全局变量不能用作可变变量。$this 变量也是一个特殊变量，不能被动态引用。

18.7 $this

- $this 变量不能被动态引用。
- $this 变量不能被赋值。

Chapter 19

Superglobals

19.1 Overview

PHP 实现了自己的外部变量机制，因此可以以简单的方式来处理表单数据，获取环境变量等，并将外部数据或变量保存到全局数组变量中。

对于全部脚本而言，PHP 提供了大量的预定义变量，不过这些预定义变量依赖于运行的服务器的版本和设置以及其它因素。

注意，一些预定义变量在 PHP 以命令行形式运行时并不生效，不过如果有可用的 PHP 预定义变量那还是最好使用（例如超全局数组等)。

预定义变量可以让 PHP 将所有的外部变量表示成内建环境变量，并且将错误信息表示成返回头。

具体来说，PHP 提供的附加的预定义数组变量包含了来自 Web 服务器、运行环境和用户输入的数据，而且这些数据的特别之处在于它们是在全局范围内自动生效的，因此通常被称为自动全局变量（autoglobals）或者超全局变量（superglobals)。

PHP 超全局变量是在全部作用域中始终可用的内置变量，包括 $GLOBALS[1]、$_SERVER、$_GET、$_POST、$_FILES、$_COOKIE、$_SESSION、$_REQUEST、$_ENV 等。

PHP 中的许多预定义变量都是“超全局的”，因此“Superglobal”也称为自动化的全局变量，它们在脚本的全部作用域中都可用，因此在函数或方法中无需执行 `global $variable;` 就可以访问它们。

19.1.1 register_globals

默认情况下，所有的超全局变量都是可用的，不过有一些指令会影响这种可用性。例如，PHP 的一个主要变化是 register_globals 指令的默认值现在已经普遍设置为 off，这个事

[1]与所有其他超全局变量不同，$GLOBALS 在 PHP 中总是可用的。

实（关闭 register_globals）将影响到预定义变量集在全局范围内的有效性。

如果已经弃用的 register_globals 指令被设置为 on 那么局部变量也将在脚本的全局作用域中可用。例如，`$_POST['foo']` 也将以 $foo 的形式存在。

- 使用 `$_SERVER['DOCUMENT_ROOT']` 代替 `$DOCUMENT_ROOT` 来得到 `DOCUMENT_ROOT` 的值。
- 使用 `$_GET['id']` 代替 `$id` 从http://www.example.com/test.php?id=3中获取 `id` 值。
- 使用 `$_ENV['HOME']` 代替 `$HOME` 获取环境变量 `HOME` 的值。

PHP 没有提供用户自定义超全局变量的机制，而且超全局变量也不能被用作函数或类方法中的可变变量。

19.1.2 register_long_arrays

PHP 的 register_long_arrays 设置选项可禁用长类型的 PHP 预定义变量数组。

超全局变量和 HTTP_*_VARS 尽管可能同时存在，但是它们并不是同一个变量，所以改变其中一个的值并不会对另一个产生影响。

对于使用 GET 或 POST 方法传递的表单数据，PHP 脚本可以通过 $_GET 或 $_POST 超全局数组变量进行保存和操作。

```
<form action="<?php echo htmlspecialchars($_SERVER['PHP_SELF'])?>" method="get">
Name: <input type="text" name="username"><br />
Email: <input type="text" name="email''></ br>
<input type="submit" name="submit" value="注册">
</form>

<pre>
您的用户名是: <?php echo $_GET['username']; ?>
Email地址是: <?php echo $_GET['email']; ?>
</pre>
```

PHP 中的超全局变量 $_GET 和 $_POST 可以自动获取表单的 HTML 数据，并且可以以数组的方式来进行操作。

- $_REQUEST 不仅包含 $_GET 和 $_POST 的值，还包括 $_COOKIE 的值。
- $_REQUEST 对 $_GET、$_POST 和 $_COOKIE 的包含是有顺序的，由 php.ini 中的 variable_order 选项决定。

例如，如果 variable_order 的值为 EGPCS，那么在 $_REQUEST 中的 GET 方法将覆盖 POST 方法的同名变量。

如果 php.ini 中的某些 variables_order 中的变量没有设定，它们的对应的 PHP 预定义数组也是空的。

Cookie 和 Session 为 PHP 提供了两种不同的会话机制，可以允许 Web 应用程序在同一个会话中自由地存取持久性会话数据。

- $_COOKIE 是客户端用来获取或存储 Cookie 的变量。
- $_SESSION 是服务器端用来获取或存储 Session 的变量。

不同的机器的环境变量是不同的，PHP 可以通过 $_ENV 和 $_SERVER 获取系统的环境变量，这些环境变量包含了 Web 服务器的相关配置信息，以及浏览器的相关状态信息，这样就可以根据不同的环境来设置不同的输出效果。

例如，从 $_SERVER 可以返回发出请求的客户机 IP 地址和服务器域名等，phpinfo() 函数返回的环境信息中就包含了 $_ENV 和 $_SERVER 的内容。

另外，在基于权限或身份认证的系统中，可以获取客户端的 IP 地址来屏蔽非法的用户请求等。

19.2 $GLOBALS

$GLOBALS 可以引用全局作用域中可用的全部变量，$GLOBALS 实际上是一个包含了全部变量的全局组合数组，其中变量的名字就是数组的键。

```
<?php
function test() {
    $foo = "local variable";

    echo '$foo in global scope: ' . $GLOBALS["foo"] . "\n";
    echo '$foo in current scope: ' . $foo . "\n";
}

$foo = "Example content";
test();
?>
```

以上例程的输出类似于：

```
$foo in global scope: Example content
$foo in current scope: local variable
```

19.3 $_SERVER

$_SERVER（服务器和执行环境信息）是一个包含了诸如头信息 (header)、路径 (path)、以及脚本位置 (script locations) 等等信息的数组。这个数组中的项目由 Web 服务器创建。

实际上，无法保证每个服务器都提供全部项目，服务器可能会忽略一些，或者提供一些没有在这里列举出来的项目，这也就意味着大量的此类变量都会在 CGI 1.1 规范中说明。

虽然 $HTTP_SERVER_VARS（已弃用）包含着相同的信息，但它不是一个超全局变量，$HTTP_SERVER_VARS 与 $_SERVER 是不同的变量，PHP 处理它们的方式不同。

如果以命令行方式运行 PHP，下面列出的元素几乎没有有效的 (或是没有任何实际意义的)。

- 'PHP_SELF' 当前执行脚本的文件名，与 document root 有关。例如，在地址为 http://example.com/test.php/f 的脚本中使用 $_SERVER['PHP_SELF'] 将得到 /test.php/foo.bar。__FILE__ 常量包含当前 (例如包含) 文件的完整路径和文件名。
  从 PHP 4.3.0 版本开始，如果 PHP 以命令行模式运行，这个变量将包含脚本名。之前的版本该变量不可用。
- 'argv' 传递给该脚本的参数的数组。当脚本以命令行方式运行时，argv 变量传递给程序 C 语言样式的命令行参数。当通过 GET 方式调用时，该变量包含 query string。
- 'argc' 包含命令行模式下传递给该脚本的参数的数目 (如果运行在命令行模式下)。
- 'GATEWAY_INTERFACE' 服务器使用的 CGI 规范的版本，例如“CGI/1.1”。
- 'SERVER_ADDR' 当前运行脚本所在的服务器的 IP 地址。
- 'SERVER_NAME' 当前运行脚本所在的服务器的主机名。如果脚本运行于虚拟主机中，该名称是由那个虚拟主机所设置的值决定。
- 'SERVER_SOFTWARE' 服务器标识字符串，在响应请求时的头信息中给出。
- 'SERVER_PROTOCOL' 请求页面时通信协议的名称和版本，例如“HTTP/1.0”。
- 'REQUEST_METHOD' 向服务器页面传送数据的方法，或者说访问页面使用的请求方法，HTTP 支持的请求方法包括“GET”、“HEAD”、“POST”和“PUT”等。
- 'REQUEST_TIME' 请求开始时的时间戳。从 PHP 5.1.0 起可用。
- 'REQUEST_TIME_FLOAT' 请求开始时的时间戳，微秒级别的精准度。自 PHP 5.4.0 开始生效。
- 'QUERY_STRING' 客户端发送到服务器的 query string（查询字符串），可以用于进行页面访问。
- 'DOCUMENT_ROOT' 当前运行脚本所在的文档根目录，例如在 Apache 服务器配置文件中定义的虚拟目录。
- 'HTTP_ACCEPT' 当前请求头中 Accept: 项的内容，如果存在的话。
- 'HTTP_ACCEPT_CHARSET' 当前请求头中 Accept-Charset: 项的内容，如果存在的话。例如：“iso-8859-1,*,utf-8”。
- 'HTTP_ACCEPT_ENCODING' 当前请求头中 Accept-Encoding: 项的内容，如果存在的话。例如：“gzip”。
- 'HTTP_ACCEPT_LANGUAGE' 当前请求头中 Accept-Language: 项的内容，如果存在的话。例如：“en”。

- 'HTTP_CONNECTION' 当前请求头中 Connection: 项的内容，例如“Keep-Alive”。
- 'HTTP_COOKIE' 当前请求头中 Cookie: 项的内容。
- 'HTTP_HOST' 当前请求头中 Host: 项的内容。
- 'HTTP_REFERER' 引导用户代理到当前页的前一页的地址（如果存在），由 user agent 设置决定。并不是所有的用户代理都会设置该项，有的还提供了修改 HTTP_REFERER 的功能，因此该值并不可信。
- 'HTTP_USER_AGENT' 当前请求头中 User-Agent: 项的字符串内容，可以表明访问该页面的用户代理的信息，例如客户端浏览器名称、版本和其他信息等。除此之外，还可以通过 get_browser() 来使用该值，从而定制页面输出以便适应用户代理的性能。
- 'HTTPS' 如果脚本是通过 HTTPS 协议被访问，则被设为一个非空的值。
- 'REMOTE_ADDR' 浏览当前页面的用户的 IP 地址。
- 'REMOTE_HOST' 浏览当前页面的用户的主机名。DNS 反向解析不依赖于用户的 REMOTE_ADDR。
- 'REMOTE_PORT' 用户机器上连接到 Web 服务器时所使用的端口号。
- 'REMOTE_USER' 经验证的用户
- 'REDIRECT_REMOTE_USER' 验证的用户，如果请求已在内部重定向。
- 'SCRIPT_FILENAME' 当前执行脚本的绝对路径。
- 'SERVER_ADMIN' Apache 服务器配置文件中的 SERVER_ADMIN 参数。如果脚本运行在一个虚拟主机上，则该值是那个虚拟主机的值。
- 'SERVER_PORT' Web 服务器等待请求的端口，默认值为“80”。如果使用 SSL 安全连接，则这个值为用户设置的 HTTP 端口。
- 'SERVER_SIGNATURE' 包含了服务器版本和虚拟主机名的字符串。
- 'PATH_TRANSLATED' 当前脚本所在文件系统（非文档根目录）的基本路径。这是在服务器进行虚拟到真实路径的映像后的结果。
- 'SCRIPT_NAME' URL 中包含当前脚本的路径。这在页面需要指向自己时非常有用。__FILE__ 常量包含当前脚本 (例如包含文件) 的完整路径和文件名。
- 'REQUEST_URI' 要访问的页面。例如“/index.html”。
- 'PHP_AUTH_DIGEST' 当作为 Apache 模块运行时，进行 HTTP Digest 认证的过程中，此变量被设置成客户端发送的“Authorization”HTTP 头内容（以便作进一步的认证操作）。
- 'PHP_AUTH_USER' 当 PHP 运行在 Apache 或 IIS（PHP 5 是 ISAPI）模块方式下，并且正在使用 HTTP 认证功能，这个变量便是用户输入的用户名。
- 'PHP_AUTH_PW' 当 PHP 运行在 Apache 或 IIS（PHP 5 是 ISAPI）模块方式下，并且正在使用 HTTP 认证功能，这个变量便是用户输入的密码。
- 'AUTH_TYPE' 当 PHP 运行在 Apache 模块方式下，并且正在使用 HTTP 认证功能，这个变量便是认证的类型。

- 'PATH_INFO' 包含由客户端提供的、跟在真实脚本名称之后并且在查询语句（query string）之前的路径信息，如果存在的话。
  例如，如果当前脚本是通过 URL http://www.example.com/php/path_info.php/some/stuff?foo=bar 被访问，那么 $_SERVER['PATH_INFO'] 将包含 /some/stuff。
- 'ORIG_PATH_INFO' 在被 PHP 处理之前，“PATH_INFO”的原始版本。

```
<?php
echo $_SERVER['SERVER_NAME'];
?>
```

以上例程的输出类似于：

```
www.example.com
```

19.4 $_GET

- $_GET 适用于 HTTP 查询字符串（query string），也就是 URL 中“?”之后的信息。
- $_GET 是通过 URL 参数传递给当前脚本的变量的数组，$_GET 默认是通过 urldecode() 传递的。

虽然 $HTTP_GET_VARS（已弃用）包含相同的信息，但它不是一个超全局变量，而且 $HTTP_GET_VARS 和 $_GET 是不同的变量，PHP 处理它们的方式是不同的。

```
<?php
echo 'Hello ' . htmlspecialchars($_GET["name"]) . '!';
?>
```

假设用户访问的是 http://example.com/?name=Hannes，以上例程的输出类似于：

```
Hello Hannes!
```

当一个表单提交给 PHP 脚本时，表单中的信息会自动在脚本中可用，而且可以有多种方法来访问 HTML 表单中的数据。

```
<form action="foo.php" method="GET">
   Name: <input type="text" name="username"><br />
   Email: <input type="text" name="email"><br />
   <input type="submit" name="submit" value="Submit me!" />
</form>
```

19.5 $_POST

$_POST 变量是通过 HTTP POST 方法传递给当前脚本的变量的数组。

虽然 $HTTP_POST_VARS（已弃用）包含相同的信息，但它不是一个超全局变量，而且 $HTTP_POST_VARS 和 $_POST 是不同的变量，PHP 处理它们的方式是不同的。

```
<?php
echo 'Hello ' . htmlspecialchars($_POST["name"]) . '!';
?>
```

假设用户通过 HTTP POST 方式传递了参数 `name=Hannes`，以上例程的输出类似于 `Hello Hannes!`。

```
<?php
// 自 PHP 4.1.0 起可用
   echo $_POST['username'];
   echo $_REQUEST['username'];

   import_request_variables('p', 'p_');
   echo $p_username;

// 自 PHP 5.0.0 起，这些长格式的预定义变量
// 可用 register_long_arrays 指令关闭。

   echo $HTTP_POST_VARS['username'];

// 如果 PHP 指令 register_globals = on 时可用。不过自
// PHP 4.2.0 起默认值为 register_globals = off。
// 不提倡使用/依赖此种方法。

   echo $username;
?>
```

使用 GET 表单的情况是类似的，而且 GET 也适用于 QUERY_STRING（URL 中在“?”之后的信息）。

变量名中的点和空格被转换成下划线。例如，`<input name="a.b" />` 变成了 `$_REQUEST["a_b"]`。

magic_quotes_gpc 配置指令影响到 $_GET，$_POST 和 $_COOKIE 的值。如果打开，那么值 (`It's "PHP!"`) 会自动转换成 (`It\'s \"PHP!\"`)，以前对数据库的插入需要这种方式进行转义，现在已经过时了，应该关闭 magic_quotes_gpc 。

另外，PHP 也懂得表单变量上下文中的数组，可以将相关的变量编成组，或者用此特性从多选输入框中取得值。例如，将一个表单 POST 给自己并在提交时显示数据：

```
<?php
```

```
if (isset($_POST['action']) && $_POST['action'] == 'submitted') {
    echo '<pre>';

    print_r($_POST);
    echo '<a href="'. $_SERVER['PHP_SELF'] .'">Please try again</a>';

    echo '</pre>';
} else {
?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
    Name: <input type="text" name="personal[name]"><br />
    Email: <input type="text" name="personal[email]"><br />
    Beer: <br>
    <select multiple name="beer[]">
        <option value="warthog">Warthog</option>
        <option value="guinness">Guinness</option>
        <option value="stuttgarter">Stuttgarter Schwabenbr</option>
    </select><br />
    <input type="hidden" name="action" value="submitted" />
    <input type="submit" name="submit" value="submit me!" />
</form>
<?php
}
?>
```

当提交表单时，可以用图片代替标准的提交按钮：

```
<input type="image" src="image.gif" name="sub" />
```

当用户点击到图像中的某处时，相应的表单会被传送到服务器，并加上两个变量 sub_x 和 sub_y，它们包含了用户点击图像的坐标。有经验的用户可能会注意到被浏览器发送的实际变量名包含的是一个点而不是下划线（即 sub.x 和 sub.y），PHP 会自动将点转换成下划线。

19.6 $_FILES

$_FILES（HTTP 文件上传变量）是通过 HTTP POST 方式上传到当前脚本的项目的数组。

虽然 $HTTP_POST_FILES（已弃用）包含相同的信息，但是它不是一个超全局变量，而且 $HTTP_POST_FILES 和 $_FILES 是不同的变量，PHP 处理它们的方式是不同的。

19.7 $_REQUEST

默认情况下，$_REQUEST（HTTP Request 变量）包含了 $_GET，$_POST 和 $_COOKIE 的数组。

由于 $_REQUEST 中的变量通过 GET，POST 和 COOKIE 输入机制传递给脚本文件，因此可以被远程用户篡改而并不可信。

$_REQUEST 数组的项目及其顺序依赖于 PHP 的 variables_order 指令的配置。

以命令行方式运行 PHP 时，$_REQUEST 将不包含 `argv` 和 `argc` 信息，它们实际上存在于 $_SERVER 数组中。

19.8 $_SESSION

$_SESSION 变量是当前脚本可用 SESSION 变量的数组。

虽然 $HTTP_SESSION_VARS（已弃用）包含相同的信息，但它不是一个超全局变量，$HTTP_SESSION_VARS 和 $_SESSION 是不同的变量，PHP 处理它们的方式是不同的。

19.9 $_ENV

$_ENV（环境变量）是通过环境方式传递给当前脚本的变量的数组。

这些变量被从 PHP 解析器的运行环境导入到 PHP 的全局命名空间。很多是由支持 PHP 运行的 Shell 提供的，并且不同的系统很可能运行着不同种类的 Shell，所以不可能有一份确定的列表。

其他环境变量包含了 CGI 变量，而不管 PHP 是以服务器模块还是 CGI 处理器的方式运行。

虽然 $HTTP_ENV_VARS(已弃用)包含相同的信息,但它不是一个超全局变量,$HTTP_ENV_VARS 和 $_ENV 是不同的变量，PHP 处理它们的方式是不同的。

```
<?php
echo 'My username is ' .$_ENV["USER"] . '!';
?>
```

假设"bjori" 运行此段脚本，以上例程的输出类似于:

```
My username is bjori!
```

19.10 $_COOKIE

$_COOKIE 变量是通过 HTTP Cookies 方式传递给当前脚本的变量的数组。

虽然 $HTTP_COOKIE_VARS（已弃用）包含相同的信息，但它不是一个超全局变量，$HTTP_COOKIE_VARS 和 $_COOKIE 是不同的变量，PHP 处理它们的方式是不同的。

```
<?php
echo 'Hello ' . htmlspecialchars($_COOKIE["name"]) . '!';
?>
```

假设之前发送了"name" Cookie，以上例程的输出类似于：

```
Hello Hannes!
```

Cookies 是一种在远端浏览器端存储数据并能追踪或识别再次访问的用户的机制，PHP 透明地支持 RFC 6265 定义中的 HTTP cookies。

Cookies 是 HTTP 信息头中的一部分，PHP 提供了 setcookie() 函数来设定 cookies，而且 setcookie 函数必须在向浏览器发送任何输出之前调用，对于 header() 函数也有同样的限制。

HTTP Cookie 数据会在相应的 COOKIE 数据数组中可用，例如 $_COOKIE，$HTTP_COOKIE_VARS 和 $_REQUEST。

如果要将多个值赋给一个 cookie 变量，必须将其赋成数组。例如，下面的示例将会建立两个单独的 cookie，尽管 MyCookie 在脚本中是一个单一的数组。如果想在仅仅一个 cookie 中设定多个值，考虑先在值上使用 serialize() 或 explode()。

```
<?php
  setcookie("MyCookie[foo]", 'Testing 1', time()+3600);
  setcookie("MyCookie[bar]", 'Testing 2', time()+3600);
?>
```

在浏览器中一个 cookie 会替换掉上一个同名的 cookie，除非路径或者域不同，因此购物车程序可以保留一个计数器并一起传递。

```
<?php
if (isset($_COOKIE['count'])) {
   $count = $_COOKIE['count'] + 1;
} else {
   $count = 1;
}
setcookie('count', $count, time()+3600);
setcookie("Cart[$count]", $item, time()+3600);
?>
```

19.11 $php_errormsg

$php_errormsg 变量包含由 PHP 生成的最新错误信息，并且仅在 php.ini 文件中的 track_errors 配置项开启的情况下可用。

$php_errormsg 变量只在错误发生的作用域内可用，并且要求 track_errors 配置项是开启的（默认是关闭的）。

如果用户定义了错误处理句柄(set_error_handler())并且返回 FALSE 的时候，$php_errormsg 就会被设置。

```
<?php
@strpos();
echo $php_errormsg;
?>
```

以上例程的输出类似于：

```
Wrong parameter count for strpos()
```

19.12 $HTTP_RAW_POST_DATA

$HTTP_RAW_POST_DATA（已弃用）是原生 POST 数据，包含 POST 提交的原始数据。

一般而言，使用 php://input 代替 $HTTP_RAW_POST_DATA。

19.13 $http_response_header

当使用 HTTP 包装器时，$http_response_header（HTTP 响应头）将会被 HTTP 响应头信息填充。

- $http_response_header 数组与 get_headers() 函数类似。
- $http_response_header 将被创建于局部作用域中。

```
<?php
function get_contents() {
  file_get_contents("http://example.com");
  var_dump($http_response_header);
}
get_contents();
var_dump($http_response_header);
?>
```

以上例程的输出类似于：

```
array(9) {
  [0]=>
  string(15) "HTTP/1.1 200 OK"
  [1]=>
  string(35) "Date: Sat, 12 Apr 2008 17:30:38 GMT"
```

```
  [2]=>
  string(29) "Server: Apache/2.2.3 (CentOS)"
  [3]=>
  string(44) "Last-Modified: Tue, 15 Nov 2005 13:24:10 GMT"
  [4]=>
  string(27) "ETag: "280100-1b6-80bfd280""
  [5]=>
  string(20) "Accept-Ranges: bytes"
  [6]=>
  string(19) "Content-Length: 438"
  [7]=>
  string(17) "Connection: close"
  [8]=>
  string(38) "Content-Type: text/html; charset=UTF-8"
}
NULL
```

19.14 $argc

$argc 包含当运行于命令行下时传递给当前脚本的参数的数目，而且这个变量仅在 register_argc_argv 打开时可用。

脚本的文件名总是作为参数传递给当前脚本，因此 $argc 的最小值为 1。

```
<?php
var_dump($argc);
?>
```

当使用这个命令执行: `php script.php arg1 arg2 arg3`，以上例程的输出类似于:

```
int(4)
```

19.15 $argv

$argv 变量包含当运行于命令行下时传递给当前脚本的参数的数组，而且这个变量仅在 register_argc_argv 打开时可用。

第一个参数总是当前脚本的文件名，因此 `$argv[0]` 就是脚本文件名。

```
<?php
var_dump($argv);
?>
```

当使用这个命令执行：`php script.php arg1 arg2 arg3`，以上例程的输出类似于：

```
array(4) {
  [0]=>
  string(10) "script.php"
  [1]=>
  string(4) "arg1"
  [2]=>
  string(4) "arg2"
  [3]=>
  string(4) "arg3"
}
```

Chapter 20

PHP Constants

常量是一个简单值的标识符（名字），在程序执行期间其值不能改变（除了所谓的魔术常量，它们其实不是常量）。

- 常量默认为大小写敏感，而且常量标识符通常总是大写的。
- 常量名和其它任何 PHP 标签遵循同样的命名规则，只是不带“$”符号。

通常情况下，合法的常量名以字母或下划线开始，后面跟着任何字母，数字或下划线。

在正则表达式可以这样描述常量名：`[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*`[1]。

常量只能包含标量数据（boolean，integer，float 和 string），虽然可以定义 resource 常量，但是应该尽量避免，否则可能造成不可预料的结果。

用户可以用 define() 函数来定义常量，也可以使用 const 关键字在类定义之外定义常量。

常量不能是数组、对象或资源，用户不能再改变常量的值或者取消定义。

```
<?php

// 合法的常量名
define("FOO",  "something");
define("FOO2", "something else");
define("FOO_BAR", "something more");

// 非法的常量名
define("2FOO", "something");

// 下面的定义是合法的，但应该避免这样做：（自定义常量不要以__开头）
// 也许将来有一天PHP会定义一个__FOO__的魔术常量
// 这样就会与你的代码相冲突
define("__FOO__", "something");
```

[1]在这里，字母指的是 a-z，A-Z，以及从 127 到 255（0x7f-0xff）的 ASCII 字符。

```
echo FOO;
echo FOO2;
echo FOO_BAR;
echo __FOO__;
?>
```

使用关键字 const 定义常量的语法如下：

```
<?php
// 以下代码在 PHP 5.3.0 后可以正常工作
const CONSTANT = 'Hello World';

echo CONSTANT;
?>
```

和使用 define() 来定义常量相反的是，使用 const 关键字定义常量必须处于最顶端的作用区域，因为 const 是在编译时定义的，同时也就意味着不能在函数内，循环内以及 if 语句内部用 const 来定义常量。

常量和变量有如下不同：

- 常量前面没有美元符号（$）；
- 常量用 define() 函数或 const 关键字[2]定义，而不能通过赋值语句；
- 常量可以在任何地方定义和访问；
- 常量定义后就不能被重新定义或者取消定义；
- 常量的值只能是标量。

可以简单的通过指定其名字来取得常量的值，与变量不同，不应该在常量前面加上 $ 符号。如果常量名是动态的，也可以用函数 constant() 来获取常量的值。用 get_defined_constants() 可以获得所有已定义的常量列表。

如果未经定义就使用一个常量，那么 PHP 将会自动为其赋值，并且 PHP 假定想要的是该常量本身的名字，如同用字符串调用它一样（CONSTANT 对应 "CONSTANT"），同时将发出一个 E_NOTICE 级的错误。

例如，除非事先用 define() 将 bar 定义为一个常量，否则 `$foo[bar]` 是错误的。

- 如果只想检查是否定义了某常量，用 defined() 函数。
- 如果只想检查是否定义了某变量，用 isset() 函数。

```
<?php
// 检查常量
if(defined("M_EULER"){
  define("M_EULER",0.5772156649015);
```

[2]const 关键字可以用于在类定义之外定义常量。

```
}

// 检查变量
if(isset($_GET['page'])){
  $_GET['page'] = 1;
}
?>
```

常量和（全局）变量在不同的名字空间中，这意味着 TRUE 和 $TRUE 就是不同的。

和 superglobals 一样，常量的范围是全局的，因此可以在脚本的任何地方访问常量。

如果只想检查是否定义了某常量，用 defined() 函数。

20.1 Magic Constants

PHP 为运行的任何脚本都提供了大量的预定义常量，其中很多常量都是由不同的扩展库定义的，只有在加载了这些扩展库时才会出现，或者动态加载后，或者在编译时已经包含。

有 8 个魔术常量它们的值随着它们在代码中的位置改变而改变（例如 __LINE__ 的值就依赖于它在脚本中所处的行），这些特殊的常量不区分大小写，如下：

表 20.1: PHP 的“魔术常量”

名称	说明
__LINE__	文件中的当前行号。
__FILE__	文件的完整路径和文件名。如果用在被包含文件中，则返回被包含的文件名。自 PHP 4.0.2 起，__FILE__ 总是包含一个绝对路径（如果是符号连接，则是解析后的绝对路径），而在此之前的版本有时会包含一个相对路径。
__DIR__	文件所在的目录。如果用在被包括文件中，则返回被包括的文件所在的目录。它等价于 dirname(__FILE__)。除非是根目录，否则目录中名不包括末尾的斜杠。
__FUNCTION__	函数名称。自 PHP 5 起本常量返回该函数被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的。
__CLASS__	类的名称。自 PHP 5 起本常量返回该类被定义时的名字（区分大小写）。在 PHP 4 中该值总是小写字母的。类名包括其被声明的作用区域（例如 Foo\Bar），而且 __CLASS__ 对 trait 也起作用。当用在 trait 方法中时，__CLASS__ 是调用 trait 方法的类的名字。

名称	说明
__TRAIT__	Trait 的名字。本常量返回 trait 被定义时的名字 (区分大小写)。Trait 名包括其被声明的作用区域（例如 Foo\Bar)。
__METHOD__	类的方法名。返回该方法被定义时的名字（区分大小写)。
__NAMESPACE__	当前命名空间的名称（区分大小写)。此常量是在编译时定义的。
TURE	布尔值，真
FALSE	布尔值，假
NULL	NULL 值
PHP_OS	操作系统类型，例如“WINNT”
PHP_VERSION	PHP 版本号
DIRECTORY_SEPARATOR	目录分隔符，UNIX/Linux 支持/，Windows 支持\
PHP_SHLIB_SUFFIX	UNIX/Linux 使用“so”，Windows 使用“dll”
PATH_SEPARATOR	环境变量 PATH 中使用的目录列表分隔符，UNIX/Linux 使用“:”，Windows 使用“;”
E_ERROR	1，表示错误
E_WARNING	2，表示警告
E_PARSE	4，表示解析错误，由程序解析器报告
E_NOTICE	8，表示非关键错误
M_PI	圆周率，3.1415926535898
M_E	自然常数，2.7182818284590
M_LOG2E	$\log_2^e$=1.4426950408890，以 2 为底的 e 的对数
M_LOG10E	$\log_{10}^e$=0.4342944819033，以 10 为底的 e 的对数
M_LN2	$\ln 2$=0.6931471805599，以 e 为底的 2 的对数
M_LN10	$\ln 10$=2.3025850929940，以 e 为底的 10 的对数
M_1_PI	$1/\pi$=0.3183098861838
M_LNPI	$\ln \pi$=1.1447298858494
M_EULER	欧拉常数，0.5772156649015

Chapter 21

PHP String

PHP 中的字符串变量用于包含字符串的值，存储并处理文本片段，而且 PHP 内置相应的字符串函数来对字符串进行灵活的操作。

PHP 字符串函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

在创建字符串之后就可以对它进行操作了，可以直接在函数中使用字符串，或者把它存储在变量中，比如在下面的例子中，PHP 脚本把字符串"Hello World" 赋值给名为 `$txt` 的字符串变量：

```
<?php
$txt="Hello World";
echo $txt;
?>
```

以上代码的输出：`Hello World`

C 语言的字符串是字符数组，PHP 将字符串作为基本数据类型进行处理，因此 PHP 的抽象级别比 C 语言更高。

对字符串的基本操作包括访问字符串中的字符等，PHP 支持在字符串变量后面使用 "{}" 操作符，并且通过字符的偏移量对其进行直接操作。

```
<?php
$txt="Hello World";
echo $txt{0}; // H
?>
```

21.1 Concatenation

在 PHP 中，只有一个字符串运算符，称为并置运算符 (.)，用于把两个字符串值连接起来。

要把两个变量连接在一起，可以使用这个点运算符 (.)：

```
<?php
  $txt1="Hello World";
  $txt2="1234";
  echo $txt1 . " " . $txt2;
?>
```

以上代码的输出：`Hello World 1234`

在上面的例子中使用了两次并置运算符，这是由于我们需要插入第三个字符串。为了分隔这两个变量，我们在 $txt1 与 $txt2 之间插入了一个空格。

21.2 String Functions

表 21.1: PHP String 函数

函数	描述	PHP
addcslashes()	在指定的字符前添加反斜杠。	4
addslashes()	在指定的预定义字符前添加反斜杠。	3
bin2hex()	把 ASCII 字符的字符串转换为十六进制值。	3
chop()	rtrim() 的别名。	3
chr()	从指定的 ASCII 值返回字符。	3
chunk_split()	把字符串分割为一连串更小的部分。	3
convert_cyr_string()	把字符由一种 Cyrillic 字符转换成另一种。	3
convert_uudecode()	对 uuencode 编码的字符串进行解码。	5
convert_uuencode()	使用 uuencode 算法对字符串进行编码。	5
count_chars()	返回字符串所用字符的信息。	4
crc32()	计算一个字符串的 32-bit CRC。	4
crypt()	单向的字符串加密法 (hashing)。	3
echo()	输出字符串。	3
explode()	把字符串打散为数组。	3
fprintf()	把格式化的字符串写到指定的输出流。	5
get_html_translation_table()	返回翻译表。	4
hebrev()	把希伯来文本从右至左的流转换为左至右的流。	3
hebrevc()	同上，同时把 (\n) 转为 。	3

函数	描述	PHP
html_entity_decode()	把 HTML 实体转换为字符。	4
htmlentities()	把字符转换为 HTML 实体。	3
htmlspecialchars_decode()	把一些预定义的 HTML 实体转换为字符。	5
htmlspecialchars()	把一些预定义的字符转换为 HTML 实体。	3
implode()	把数组元素组合为一个字符串。	3
join()	implode() 的别名。	3
levenshtein()	返回两个字符串之间的 Levenshtein 距离。	3
localeconv()	返回包含本地数字及货币信息格式的数组。	4
ltrim()	从字符串左侧删除空格或其他预定义字符。	3
md5()	计算字符串的 MD5 散列。	3
md5_file()	计算文件的 MD5 散列。	4
metaphone()	计算字符串的 metaphone 键。	4
money_format()	把字符串格式化为货币字符串。	4
nl_langinfo()	返回指定的本地信息。	4
nl2br()	在字符串中的每个新行之前插入 HTML 换行符。	3
number_format()	通过千位分组来格式化数字。	3
ord()	返回字符串第一个字符的 ASCII 值。	3
parse_str()	把查询字符串解析到变量中。	3
print()	输出一个或多个字符串。	3
printf()	输出格式化的字符串。	3
quoted_printable_decode()	解码 quoted-printable 字符串。	3
quotemeta()	在字符串中某些预定义的字符前添加反斜杠。	3
rtrim()	从字符串的末端开始删除空白字符或其他预定义字符。	3
setlocale()	设置地区信息（地域信息）。	3
sha1()	计算字符串的 SHA-1 散列。	4
sha1_file()	计算文件的 SHA-1 散列。	4
similar_text()	计算两个字符串的匹配字符的数目。	3
soundex()	计算字符串的 soundex 键。	3
sprintf()	把格式化的字符串写写入一个变量中。	3
sscanf()	根据指定的格式解析来自一个字符串的输入。	4

函数	描述	PHP
str_ireplace()	替换字符串中的一些字符。 (对大小写不敏感)	5
str_pad()	把字符串填充为新的长度。	4
str_repeat()	把字符串重复指定的次数。	4
str_replace()	替换字符串中的一些字符。 (对大小写敏感)	3
str_rot13()	对字符串执行 ROT13 编码。	4
str_shuffle()	随机地打乱字符串中的所有字符。	4
str_split()	把字符串分割到数组中。	5
str_word_count()	计算字符串中的单词数。	4
strcasecmp()	比较两个字符串。 (对大小写不敏感)	3
strchr()	搜索字符串在另一字符串中的第一次出现。 strstr() 的别名	3
strcmp()	比较两个字符串。 (对大小写敏感)	3
strcoll()	比较两个字符串（根据本地设置)。	4
strcspn()	返回在找到任何指定的字符之前, 在字符串查找的字符数。	3
strip_tags()	剥去 HTML、XML 以及 PHP 的标签。	3
stripcslashes()	删除由 addcslashes() 函数添加的反斜杠。	4
stripslashes()	删除由 addslashes() 函数添加的反斜杠。	3
stripos()	返回字符串在另一字符串中第一次出现的位置。 (大小写不敏感)	5
stristr()	查找字符串在另一字符串中第一次出现的位置。 (大小写不敏感)	3
strlen()	返回字符串的长度。	3
strnatcasecmp()	使用一种“自然”算法来比较两个字符串。 (对大小写不敏感)	4
strnatcmp()	使用一种“自然”算法来比较两个字符串。 (对大小写敏感)	4

函数	描述	PHP
strncasecmp()	前 n 个字符的字符串比较。 (对大小写不敏感)。	4
strncmp()	前 n 个字符的字符串比较。 (对大小写敏感)。	4
strpbrk()	在字符串中搜索指定字符中的任意一个。	5
strpos()	返回字符串在另一字符串中首次出现的位置。 (对大小写敏感)	3
strrchr()	查找字符串在另一个字符串中最后一次出现的位置。	3
strrev()	反转字符串。	3
strripos()	查找字符串在另一字符串中最后出现的位置。 (对大小写不敏感)	5
strrpos()	查找字符串在另一字符串中最后出现的位置。 (对大小写敏感)	3
strspn()	返回在字符串中包含的特定字符的数目。	3
strstr()	搜索字符串在另一字符串中的首次出现。 (对大小写敏感)	3
strtok()	把字符串分割为更小的字符串。	3
strtolower()	把字符串转换为小写。	3
strtoupper()	把字符串转换为大写。	3
strtr()	转换字符串中特定的字符。	3
substr()	返回字符串的一部分。	3
substr_compare()	从指定的开始长度比较两个字符串。	5
substr_count()	计算子串在字符串中出现的次数。	4
substr_replace()	把字符串的一部分替换为另一个字符串。	4
trim()	从字符串的两端删除空白字符和其他预定义字符。	3
ucfirst()	把字符串中的首字符转换为大写。	3
ucwords()	把字符串中每个单词的首字符转换为大写。	3
vfprintf()	把格式化的字符串写到指定的输出流。	5
vprintf()	输出格式化的字符串。	4
vsprintf()	把格式化字符串写入变量中。	4
wordwrap()	按照指定长度对字符串进行折行处理。	4

21.3 Analyze

21.3.1 strlen()

strlen() 函数用于计算字符串的长度，下面的示例中使用 strlen() 来计算出字符串”Hello world!” 的长度：

```
<?php
  echo strlen("Hello world!");
?>
```

以上代码的输出：12

字符串的长度信息常用在循环或其他函数中，因此确定字符串何时结束是很重要的（例如，在循环中，我们需要在字符串中的最后一个字符之后结束循环）。

和 C 语言不同，PHP 不会在字符串末尾添加一个“\0”作为结束标志。

21.3.2 strpos()

strpos() 函数用于在字符串内检索一段字符串或一个字符。如果在字符串中找到匹配，该函数会返回第一个匹配的位置。如果未找到匹配，则返回 FALSE。

```
mixed strpos ( string $haystack , mixed $needle [, int $offset = 0 ] )
```

下面的示例演示如何在字符串中找到子字符串”world”：

```
<?php
  echo strpos("Hello world!", "world");
?>
```

以上代码的输出是：6

在字符串”Hello world!” 中，字符串”world” 的位置是 6，至于返回 6 而不是 7，是由于字符串中的首个位置是 0，而不是 1。

21.3.3 strrpos()

```
int strrpos ( string $haystack , string $needle [, int $offset = 0 ] )
```

在使用 strpos() 和 strrpos() 函数时需要注意，如果没有查询到特定字符串出现的位置，函数将返回 FALSE，因此在判断一个字符串中是否包含另一个子串的正确方法是使用恒等符“===”。

```
if(strpos($string,"w") == false){
  // ...
}
```

21.3.4 substr()

substr() 函数用于从某一个字符串中取出子字符串。

```
string substr ( string $string , int $start [, int $length ] )
```

21.3.5 explode()

```
array explode ( string $delimiter , string $string [, int $limit ] )
```

21.3.6 implode()

```
string implode ( string $glue , array $pieces )
```

21.4 Blank

21.4.1 trim()

```
string trim ( string $str [, string $charlist = " \t\n\r\0\x0B" ] )
```

在实践中容易出现空白的地方是服务器接受的表单数据，例如用户无意中输入的空格等。

```
<?php
if(isset($_POST[''submit'])){
  if(!trim($_POST['title'])){
    echo "<p><font color=red>标题没有填写</font></p>";
  }elseif(!trim($_POST['content'])){
    echo "<p><font color=red>正文没有填写</font></p>";
  }else{
    echo $_POST['title'];
    echo "<hr>";
    echo $_POST['content'];
  }
}
<form method="post" action="#">
标题：<input type="text" name="title" value=""><br />
内容：<textarea name="content"></textarea><br />
<input type="submit" name="submit" value="提交">
</form>
?>
```

trim()、ltrim() 和 rtrim() 等函数只能删除单字节的空白字符，无法删除“全角空格”等字符。

另外，为了删除某些非打印字符产生的空白字符，可以在删除列表中指定要删除的空白字符。

21.4.2 str_pad()

```
string str_pad ( string $input , int $pad_length [, string $pad_string = " " [, int
    $pad_type = STR_PAD_RIGHT ]] )
```

str_pad() 等字符填充函数可以用于实现对敏感信息的保护，例如金额的打印操作等，其原理类似于 C 语言中的 printf() 函数。

21.5 Reverse

21.5.1 strrev()

strrev() 函数的功能类似于对字符串中的字符的反序排列，它们都是按照字节进行的。

```
string strrev ( string $string )
```

21.6 Transform

21.6.1 strtoupper()

```
string strtoupper ( string $string )
```

21.6.2 strtolower()

```
string strtolower ( string $string )
```

21.6.3 ucfirst()

```
string ucfirst ( string $str )
```

21.6.4 ucwords()

```
string ucwords ( string $str )
```

21.7 Comparison

PHP 支持使用多种方法对字符串进行比较，并且除了可以直接使用条件运算符进行比较之外，还可以按照字节、自然排序以及模糊比较等。

在使用条件运算符来比较字符串和数字时，字符串首先被转换为数字，然后再进行比较。

21.7.1 strcmp()

strcmp() 按字节进行字符串的比较。

```
int strcmp ( string $str1 , string $str2 )
```

21.7.2 strcasecmp()

strcasecmp() 函数可以忽略大小写来对字符串进行比较。

```
int strcasecmp ( string $str1 , string $str2 )
```

21.7.3 strnatcmp()

除了按照字节位进行比较之外，PHP 也可以按照“自然排序”法来比较字符串。

具体来说，自然排序法将比照字符串中的数字部分，将字符串按照数字大小进行排序，从而可以直接模拟人类的思维习惯。

```
int strnatcmp ( string $str1 , string $str2 )
```

21.7.4 strnatcasecmp()

```
int strnatcasecmp ( string $str1 , string $str2 )
```

21.7.5 soundex()

在模糊查询中，计算字符串（主要是英文单词）的相似读音可以对某一单词计算出一个关键值，这样就可以进一步找出和输入的单词有相同发音关键值的所有单词。

```
string soundex ( string $str )
```

21.7.6 similar_text()

similar_text() 可以计算两个单词的相似程度，并且返回它们中相匹配的字符的数目，这样就可以实现相似度的输出。

```
int similar_text ( string $first , string $second [, float &$percent ] )
```

21.8 Replace

21.8.1 str_replace()

str_replace() 函数和 strtr() 函数都可以实现字符替换功能，而且都可以将给定字符串的指定部分替换为其他字符串。

```
mixed str_replace ( mixed $search , mixed $replace , mixed $subject [, int &$count ] )
```

21.8.2 strtr()

strtr() 函数也称为字符串翻译函数，可以用来转换指定字符，并且返回相应的副本。

```
string strtr ( string $str , string $from , string $to )
string strtr ( string $str , array $replace_pairs )
```

21.9 Format

PHP 提供了和 C 语言同名的 printf() 函数和 sprintf() 函数来对数字或字符串进行格式化输出。

- printf() 函数直接输出格式化后的字符串。
- sprintf() 函数返回格式化后的字符串。

21.9.1 printf()

```
int printf ( string $format [, mixed $args [, mixed $... ]] )
```

21.9.2 sprintf()

```
string sprintf ( string $format [, mixed $args [, mixed $... ]] )
```

21.9.3 number_format()

```
string number_format ( float $number [, int $decimals = 0 ] )
string number_format ( float $number , int $decimals = 0 , string $dec_point = "." ,
    string $thousands_sep = "," )
```

21.10 HTML

通常情况下，在实际 Web 应用程序中接收到表单数据后需要进行立即显示或其他操作，这样可以让用户自行决定下一步是继续还是返回。

为了对 HTML 中的实体进行转义操作，可以使用 htmlentities() 函数来将所有的非 ASCII 代码转换为对应的实体代码。

21.10.1 htmlentities()

```
string htmlentities ( string $string [, int $flags = ENT_COMPAT | ENT_HTML401 [, string
    $encoding = ini_get("default_charset") [, bool $double_encode = true ]]] )
```

21.10.2 htmlspecialchars()

htmlspecialchars() 函数具有和 htmlentities() 函数相同的参数列表，其对引号的转换规则也相同。

```
string htmlspecialchars ( string $string [, int $flags = ENT_COMPAT | ENT_HTML401 [,
    string $encoding = ini_get("default_charset") [, bool $double_encode = true ]]] )
```

21.10.3 get_magic_quotes_gpc()

```
bool get_magic_quotes_gpc ( void )
```

PHP 对反斜线的处理依赖于 magic_quotes_gpc 选项的设置，因此在表单中处理数据时需要智能判定。

```
<?php
define("MAGIC_QUOTES_GPC", get_magic_quotes_gpc());

function html2text($html){
  if(MAGIC_QUOTES_GPC){
    $str = stripslashes($html);
```

```
  }
  return htmlspecialchars($str);
}
?>
```

21.10.4 stripslashes()

```
string stripslashes ( string $str )
```

21.10.5 addslashes()

```
string addslashes ( string $str )
```

21.10.6 strip_tags()

在某些场合（例如论坛）中，需要直接删除用户输入的 HTML 标签，并且使用 BBcode 等进行代替。

```
string strip_tags ( string $str [, string $allowable_tags ] )
```

21.11 URL

PHP 对 URL 的处理包括对 URL 字符串的解析、编码和构造查询字符串等。

21.11.1 parse_str()

parse_str() 函数可以使用两种不同的方式来解析浏览器使用 GET 方法发出的 QUERY_STRING 字符串，而且解析后的变量的作用域都和 parse_str() 函数相同。

- 将 QUERY_STRING 查询字符串中的变量直接转换为同名的 PHP 变量；
- 将解析后的变量放入指定的数组中。

```
mixed parse_url ( string $url [, int $component = -1 ] )
```

在下面的示例中，假设浏览器向服务器发出的 URL 是http://localhost/test.php?action=test&foo[]=hello&foo[]=world，因此 $_SERVER["QUERY_STRING"] 为 action=test&foo[]=hel

如果以第一种方式来解析 URL，作用域在函数内部。

```
<?php
function test(){
  parse_str($_SERVER["QUERY_STRING"]);
```

```
  echo $action . "<br />";
  echo $foo[1] . "<br />";
  echo $foo[2];
}
?>
```

如果以第二种方式来解析 URL，将会把变量解析到给定的数组中。

```
<?php
function test(){
  parse_str($_SERVER["QUERY_STRING"], $arr);
  echo $arr['action'] . "<br />";
  echo $arr['foo'][0] . "<br />";
  echo $arr['foo'][1];
}
?>
```

21.11.2 parse_url()

parse_url() 函数可以用来对 URL 进行全面解析，并且返回一个数组，其中包含以下部分或者全部内容。

- schema：协议，例如 http 或 https；
- host：域名；
- port：端口，例如 80；
- user：认证用户名；
- pass：认证密码；
- path：访问路径；
- query：查询字符串，即 URL 中“?”后面的内容，或者说 $_SERVER[”QUERY_STRING”]；
- fragment：锚，URL 中“#”后面的内容，也就是页面的定位标记。

parse_url() 返回的信息不能确定 URL 中的字符串是合法的，任何类似 URL 的字符串都可以被 parse_url() 函数解析。

21.11.3 rawurlencode()

为了生成符合编码规范的 URL，在为 URL 中的普通字符（非间隔符）编码时需要满足下面的规定：

> 除“-”、“_”、“.”以及英文字母和数字外，其他的任何字符均将转换为由“%”和两个十六进制数字表示的转义实体。

```
string rawurlencode ( string $str )
```

21.11.4 urlencode()

在早期的 URL 编码规范中，空格被转换为“+”，可以使用 urlencode() 函数来进行兼容旧有的编码规范。

```
string urlencode ( string $str )
```

21.11.5 rawurldecode()

21.11.6 urldecode()

21.11.7 http_build_query()

PHP5 提供的 http_build_query() 函数可以用来构造 QUERY_STRING 查询字符串。

```
string http_build_query ( mixed $query_data [, string $numeric_prefix [, string
    $arg_separator [, int $enc_type = PHP_QUERY_RFC1738 ]]] )
```

在下面的示例中，自定义的 build_query() 函数通过递归算法来模拟了 http_build_query() 函数的行为，这里没有考虑 $numeric_prefix 参数的问题。

```
<?php
function build_query($formdata, $key=null){
  $res = array();
  foreach((array)$formdata as $k=>$v){
    $tmp_key = urlencode($k);
    if($key) $tmp_key = $key . '[' . $tmp_key . ']';
    if(is_array($v)){
      $res[] = build_query($v,$tmp_key);
    }else{
      $rtes[] = $tmp_key . "=" . urlencode($v);
    }
  }
  return implode("v",$res);
}
?>
```

21.12 PHP String Constants

表 21.2: PHP String 常量

常量	描述	PHP
CRYPT_SALT_LENGTH	包含系统默认加密方法的长度。 对于标准 DES 加密，长度是 2。	
CRYPT_STD_DES	如果支持 2 字符 salt 的 DES 加密，则设置为 1，否则为 0。	
CRYPT_EXT_DES	如果支持 9 字符 salt 的 DES 加密，则设置为 1，否则为 0。	
CRYPT_MD5	如果支持以 1 开始的 12 字符 salt 的 MD5 加密，则设置为 1，否则为 0。	
CRYPT_BLOWFISH	如果支持以 2 或 2a 开始的 16 字符 salt 的 Blowfish 加密，则设置为 1，否则为 0。	
HTML_SPECIALCHARS		
HTML_ENTITIES		
ENT_COMPAT		
ENT_QUOTES		
ENT_NOQUOTES		
CHAR_MAX		
LC_CTYPE		
LC_NUMERIC		
LC_TIME		
LC_COLLATE		
LC_MONETARY		
LC_ALL		
LC_MESSAGES		
STR_PAD_LEFT		
STR_PAD_RIGHT		
STR_PAD_BOTH		

Chapter 22

PHP Operators

运算符是可以通过给出的一或多个值（用编程行话来说，表达式）来产生另一个值（因而整个结构成为一个表达式）的东西。PHP 的运算符包括`+ - * / > < >= <=` 等，与 C++ 十分类似。

PHP 运算符可按照其能接受几个值来分组。

- 一元运算符只能接受一个值，例如`!`（逻辑取反运算符）或 `++`（递增运算符）。
- 二元运算符可接受两个值，例如算术运算符 `+`（加）和 `-`（减）。
- 最后是唯一的三元运算符`?:`，可接受三个值，通常就称为“三元运算符”(尽管称为条件运算符更合适)。

PHP 运算符可以按照功能划分为算术运算符、字符串运算符、赋值运算符、位运算符、条件运算符和逻辑运算符等，而且 PHP 运算符还支持相应的优先级。

PHP 运算符优先级和结合方向控制着在表达式包含有若干个不同运算符时究竟怎样对其求值，其中运算符优先级指定了两个表达式绑定得有多“紧密”。例如，表达式 `1 + 5 * 3` 的结果是 16 而不是 18 是因为乘号（“`*`”）的优先级比加号（“`+`”）高。必要时可以用括号来强制改变优先级。如果运算符优先级相同，其结合方向决定着应该从右向左求值，还是从左向右求值。

下表按照优先级从高到低列出了运算符。同一行中的运算符具有相同优先级，此时它们的结合方向决定求值顺序。

表 22.1: PHP 运算符优先级

结合方向	运算符	附加信息
无	`clone new`	clone 和 new
左	`[`	array()

结合方向	运算符	附加信息
右	++ -- ~ (int) (float) (string) (array) (object) (bool) @	类型和递增/递减
无	instanceof	类型
右	!	逻辑运算符
左	* / %	算术运算符
左	+ - .	算术运算符和字符串运算符
左	<< >>	位运算符
无	== != === !== <>	比较运算符
左	&	位运算符和引用
左	^	位运算符
左	|	位运算符
左	&&	逻辑运算符
左	||	逻辑运算符
左	? :	三元运算符（ternary）
右	= += -= *= /= .= %= &= |= ^= <<= >>= =>	赋值运算符
左	and	逻辑运算符
左	xor	逻辑运算符
左	or	逻辑运算符
左	,	

对具有相同优先级的运算符，左结合方向意味着将从左向右求值，右结合方向则反之。对于无结合方向具有相同优先级的运算符，该运算符有可能无法与其自身结合。举例说，在 PHP 中 `1 < 2 > 1` 是一个非法语句，而 `1 <= 1 == 1` 则不是。因为 T_IS_EQUAL 运算符的优先级比 T_IS_SMALLER_OR_EQUAL 的运算符要低。

```
<?php
$a = 3 * 3 % 5; // (3 * 3) % 5 = 4
$a = true ? 0 : true ? 1 : 2; // (true ? 0 : true) ? 1 : 2 = 2

$a = 1;
$b = 2;
$a = $b += 3; // $a = ($b += 3) -> $a = 5, $b = 5
```

```
// mixing ++ and + produces undefined behavior
$a = 1;
echo ++$a + $a++; // may print 4 or 5
?>
```

使用括号（即使在并不严格需要时）通常都可以增强代码的可读性。尽管 = 比其它大多数的运算符的优先级低，PHP 仍旧允许类似如下的表达式：`if (!$a = foo())`，在此例中 foo() 的返回值被赋给了 $a。

22.1 Arithmetic Operators

表 22.2: PHP 算术运算符

运算符	说明	示例	结果
-	Negate	-$a	$a 的负值。
+	Addition	$a + $b	$a 和 $b 的和。
-	Subtraction	$a - $b	$a 和 $b 的差。
*	Multiplication	$a * $b	$a 和 $b 的积。
/	Division	$a / $b	$a 除以 $b 的商。
%	Modulus (division remainder)	$a % $b	$a 除以 $b 的余数。

- 除法运算符总是返回浮点数，只有在两个操作数都是整数（或字符串转换成的整数）并且正好能整除时才返回一个整数。
- 除法和取模运算符的除数部分不能为 0，而且取模运算符的操作数在运算之前都会转换成整数（除去小数部分）。
- 取模运算符% 的结果和被除数的符号（正负号）相同，即 `$a % $b` 的结果和 $a 的符号相同。

```
<?php
echo (5 % 3)."\n";      // prints 2
echo (5 % -3)."\n";     // prints 2
echo (-5 % 3)."\n";     // prints -2
echo (-5 % -3)."\n";    // prints -2
?>
```

取模运算符可以扩展来进行时间、日期等不同基数的数字问题进行计算机，因此可以在直接使用 PHP 的日期时间函数之外，为解决类似“某年某月某日后第多少天”之类的问题提供了新的解决方案。

```
<?php
// 早上9点
$morning =9;
/* 计算8小时后的时间 */
$afternoon = ($morning + 8) % 12; // 下午5点
?>
```

22.2 Assignment Operators

PHP 基本的赋值运算符是“=”，但是它并不是“等于”，其意味着把右边表达式的值赋给（或者拷贝给）左边的运算数，这样就不会影响右边表达式原来的值。

赋值运算表达式的值也就是所赋的值，也就是说“$a = 3”的值就是 3。

```
<?php
$a = ($b = 4) + 5; // $a 现在成了 9, 而 $b 成了 4。
?>
```

对于数组 array，对有名字的键赋值是用“=>”运算符，该运算符的优先级和其它赋值运算符相同。

在基本赋值运算符之外，还有适合于所有二元算术，数组集合和字符串运算符的“组合运算符”，这样可以在一个表达式中使用它的值并把表达式的结果赋给它。

```
<?php
$a = 3;
$a += 5; // sets $a to 8, as if we had said: $a = $a + 5;
$b = "Hello ";
$b .= "There!"; // sets $b to "Hello There!", just like $b = $b . "There!";
?>
```

赋值运算将原变量的值拷贝到新变量中（传值赋值），所以改变其中一个并不影响另一个，也适合于在密集循环中拷贝一些值（例如大数组）。

在 PHP 中普通的传值赋值行为有个例外就是碰到对象 object 时，在 PHP 5 中是以引用赋值的，除非明确使用了 clone 关键字来拷贝。

PHP 支持引用赋值，使用“$var = &$othervar;”语法。引用赋值意味着两个变量指向了同一个数据，过程中没有拷贝任何东西。

```
<?php
$a = 3;
$b = &$a; // $b 是 $a 的引用

print "$a\n"; // 输出 3
```

```
print "$b\n"; // 输出 3

$a = 4; // 修改 $a

print "$a\n"; // 输出 4
print "$b\n"; // 也输出 4, 因为 $b 是 $a 的引用, 因此也被改变
?>
```

自 PHP 5 起，new 运算符自动返回一个引用，因此再对 new 的结果进行引用赋值会发出一条 E_DEPRECATED 错误信息，之前版本会发出一条 E_STRICT 错误信息。

例如，以下代码将产生警告：

```
<?php
class C {}

/* The following line generates the following error message:
 * Deprecated: Assigning the return value of new by reference is deprecated in...
 */
$o = &new C;
?>
```

表 22.3: PHP 赋值运算符

运算符	说明	示例	结果
=	x=y	x=y	
+=	x+=y	x=x+y	
-=	x-=y	x=x-y	
*=	x*=y	x=x*y	
/=	x/=y	x=x/y	
.=	x.=y	x=x.y	
%=	x%=y	x=x%y	

22.3 Bitwise Operators

二进制计算机中的所有数字、字母或符号都是以二进制形式存储的，通过 PHP 的位运算符允许对整型数中指定的位进行求值和操作。

表 22.4: PHP 位运算符

例子	名称	结果
$a & $b	AND(按位与)	将把 $a 和 $b 中都为 1 的位设为 1。
$a | $b	OR(按位同或)	将把 $a 和 $b 中任何一个为 1 的位设为 1。
$a ^ $b	XOR(按位异或)	将把 $a 和 $b 中一个为 1 另一个为 0 的位设为 1。
~ $a	NOT(按位取反)	将 $a 中为 0 的位设为 1，反之亦然。
$a << $b	Shift left(左移)	将 $a 中的位向左移动 $b 次（每一次移动都表示“乘以 2”）。
$a >> $b	Shift right(右移)	将 $a 中的位向右移动 $b 次（每一次移动都表示“除以 2”）。

位运算是基于系统底层的运算，而且移位本身是数学运算，向任何方向移出去的位都被丢弃。

- 左移时右侧以零填充，符号位被移走意味着正负号不被保留。
- 右移时左侧以符号位填充，意味着正负号被保留。

要用括号确保想要的优先级。例如，`$a & $b == true` 先进行比较再进行按位与，`($a & $b) == true` 则先进行按位与再进行比较。

22.3.1 Errorlevel

要注意数据类型的转换，如果左右参数都是字符串，则位运算符将对字符的 ASCII 值进行操作。

```
PHP 的 ini 设定 error_reporting 使用了按位的值，
提供了关闭某个位的真实例子。要显示除了提示级别
之外的所有错误，php.ini 中是这样用的：
E_ALL & ~E_NOTICE

具体运作方式是先取得 E_ALL 的值：
00000000000000000111011111111111
再取得 E_NOTICE 的值：
00000000000000000000000000001000
然后通过 ~ 将其取反：
11111111111111111111111111110111
最后再用按位与 AND（&）得到两个值中都设定了（为 1）的位：
00000000000000000111011111110111

另外一个方法是用按位异或 XOR（^）来取得只在
```

其中一个值中设定了的位：
E_ALL ^ E_NOTICE

error_reporting 也可用来演示怎样置位。只显示错误和可恢复错误的方法是：
E_ERROR | E_RECOVERABLE_ERROR

也就是将 E_ERROR
00000000000000000000000000000001
和 E_RECOVERABLE_ERROR
00000000000000000001000000000000
用按位或 OR（|）运算符来取得在任何一个值中被置位的结果：
00000000000000000001000000000001

```
<?php
/*
 * Ignore the top section,
 * it is just formatting to make output clearer.
 */

$format = '(%1$2d = %1$04b) = (%2$2d = %2$04b)'
       . ' %3$s (%4$2d = %4$04b)' . "\n";

echo <<<EOH
 ---------   --------- -- ---------
 result      value     op test
 ---------   --------- -- ---------
EOH;


/*
 * Here are the examples.
 */

$values = array(0, 1, 2, 4, 8);
$test = 1 + 4;

echo "\n Bitwise AND \n";
foreach ($values as $value) {
   $result = $value & $test;
```

```
    printf($format, $result, $value, '&', $test);
}

echo "\n Bitwise Inclusive OR \n";
foreach ($values as $value) {
    $result = $value | $test;
    printf($format, $result, $value, '|', $test);
}

echo "\n Bitwise Exclusive OR (XOR) \n";
foreach ($values as $value) {
    $result = $value ^ $test;
    printf($format, $result, $value, '^', $test);
}
?>
```

以上例程会输出：

```
---------     ---------  -- ---------
 result         value      op test
 ---------     ---------  -- ---------
 Bitwise AND
( 0 = 0000) = ( 0 = 0000) & ( 5 = 0101)
( 1 = 0001) = ( 1 = 0001) & ( 5 = 0101)
( 0 = 0000) = ( 2 = 0010) & ( 5 = 0101)
( 4 = 0100) = ( 4 = 0100) & ( 5 = 0101)
( 0 = 0000) = ( 8 = 1000) & ( 5 = 0101)

 Bitwise Inclusive OR
( 5 = 0101) = ( 0 = 0000) | ( 5 = 0101)
( 5 = 0101) = ( 1 = 0001) | ( 5 = 0101)
( 7 = 0111) = ( 2 = 0010) | ( 5 = 0101)
( 5 = 0101) = ( 4 = 0100) | ( 5 = 0101)
(13 = 1101) = ( 8 = 1000) | ( 5 = 0101)

 Bitwise Exclusive OR (XOR)
( 5 = 0101) = ( 0 = 0000) ^ ( 5 = 0101)
( 4 = 0100) = ( 1 = 0001) ^ ( 5 = 0101)
( 7 = 0111) = ( 2 = 0010) ^ ( 5 = 0101)
```

```
( 1 = 0001) = ( 4 = 0100) ^ ( 5 = 0101)
(13 = 1101) = ( 8 = 1000) ^ ( 5 = 0101)
```

```
<?php
echo 12 ^ 9; // Outputs '5'

echo "12" ^ "9"; // Outputs the Backspace character (ascii 8)
              // ('1' (ascii 49)) ^ ('9' (ascii 57)) = #8

echo "hallo" ^ "hello"; // Outputs the ascii values #0 #4 #0 #0 #0
                    // 'a' ^ 'e' = #4

echo 2 ^ "3"; // Outputs 1
            // 2 ^ ((int)"3") == 1

echo "2" ^ 3; // Outputs 1
            // ((int)"2") ^ 3 == 1
?>
```

22.3.2 Integershift

下面的示例说明了整数的位移:

```
<?php
/*
 * Here are the examples.
 */

echo "\n--- BIT SHIFT RIGHT ON POSITIVE INTEGERS ---\n";

$val = 4;
$places = 1;
$res = $val >> $places;
p($res, $val, '>>', $places, 'copy of sign bit shifted into left side');

$val = 4;
$places = 2;
$res = $val >> $places;
p($res, $val, '>>', $places);

$val = 4;
```

```
$places = 3;
$res = $val >> $places;
p($res, $val, '>>', $places, 'bits shift out right side');

$val = 4;
$places = 4;
$res = $val >> $places;
p($res, $val, '>>', $places, 'same result as above; can not shift beyond 0');


echo "\n--- BIT SHIFT RIGHT ON NEGATIVE INTEGERS ---\n";

$val = -4;
$places = 1;
$res = $val >> $places;
p($res, $val, '>>', $places, 'copy of sign bit shifted into left side');

$val = -4;
$places = 2;
$res = $val >> $places;
p($res, $val, '>>', $places, 'bits shift out right side');

$val = -4;
$places = 3;
$res = $val >> $places;
p($res, $val, '>>', $places, 'same result as above; can not shift beyond -1');


echo "\n--- BIT SHIFT LEFT ON POSITIVE INTEGERS ---\n";

$val = 4;
$places = 1;
$res = $val << $places;
p($res, $val, '<<', $places, 'zeros fill in right side');

$val = 4;
$places = (PHP_INT_SIZE * 8) - 4;
$res = $val << $places;
p($res, $val, '<<', $places);

$val = 4;
```

```
$places = (PHP_INT_SIZE * 8) - 3;
$res = $val << $places;
p($res, $val, '<<', $places, 'sign bits get shifted out');

$val = 4;
$places = (PHP_INT_SIZE * 8) - 2;
$res = $val << $places;
p($res, $val, '<<', $places, 'bits shift out left side');


echo "\n--- BIT SHIFT LEFT ON NEGATIVE INTEGERS ---\n";

$val = -4;
$places = 1;
$res = $val << $places;
p($res, $val, '<<', $places, 'zeros fill in right side');

$val = -4;
$places = (PHP_INT_SIZE * 8) - 3;
$res = $val << $places;
p($res, $val, '<<', $places);

$val = -4;
$places = (PHP_INT_SIZE * 8) - 2;
$res = $val << $places;
p($res, $val, '<<', $places, 'bits shift out left side, including sign bit');


/*
 * Ignore this bottom section,
 * it is just formatting to make output clearer.
 */

function p($res, $val, $op, $places, $note = '') {
    $format = '%0' . (PHP_INT_SIZE * 8) . "b\n";

    printf("Expression: %d = %d %s %d\n", $res, $val, $op, $places);

    echo " Decimal:\n";
    printf(" val=%d\n", $val);
    printf(" res=%d\n", $res);
```

```
    echo " Binary:\n";
    printf(' val=' . $format, $val);
    printf(' res=' . $format, $res);

    if ($note) {
        echo " NOTE: $note\n";
    }

    echo "\n";
}
?>
```

以上例程在 32 位机器上的输出如下：

```
--- BIT SHIFT RIGHT ON POSITIVE INTEGERS ---
Expression: 2 = 4 >> 1
 Decimal:
  val=4
  res=2
 Binary:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000010
 NOTE: copy of sign bit shifted into left side

Expression: 1 = 4 >> 2
 Decimal:
  val=4
  res=1
 Binary:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000001

Expression: 0 = 4 >> 3
 Decimal:
  val=4
  res=0
 Binary:
  val=00000000000000000000000000000100
```

```
  res=00000000000000000000000000000000
 NOTE: bits shift out right side

Expression: 0 = 4 >> 4
 Decimal:
  val=4
  res=0
 Binary:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000000
 NOTE: same result as above; can not shift beyond 0


--- BIT SHIFT RIGHT ON NEGATIVE INTEGERS ---
Expression: -2 = -4 >> 1
 Decimal:
  val=-4
  res=-2
 Binary:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111110
 NOTE: copy of sign bit shifted into left side

Expression: -1 = -4 >> 2
 Decimal:
  val=-4
  res=-1
 Binary:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111111
 NOTE: bits shift out right side

Expression: -1 = -4 >> 3
 Decimal:
  val=-4
  res=-1
```

```
 Binary:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111111
 NOTE: same result as above; can not shift beyond -1


--- BIT SHIFT LEFT ON POSITIVE INTEGERS ---
Expression: 8 = 4 << 1
 Decimal:
  val=4
  res=8
 Binary:
  val=00000000000000000000000000000100
  res=00000000000000000000000000001000
 NOTE: zeros fill in right side

Expression: 1073741824 = 4 << 28
 Decimal:
  val=4
  res=1073741824
 Binary:
  val=00000000000000000000000000000100
  res=01000000000000000000000000000000

Expression: -2147483648 = 4 << 29
 Decimal:
  val=4
  res=-2147483648
 Binary:
  val=00000000000000000000000000000100
  res=10000000000000000000000000000000
 NOTE: sign bits get shifted out

Expression: 0 = 4 << 30
 Decimal:
  val=4
```

```
  res=0
 Binary:
  val=00000000000000000000000000000100
  res=00000000000000000000000000000000
 NOTE: bits shift out left side


--- BIT SHIFT LEFT ON NEGATIVE INTEGERS ---
Expression: -8 = -4 << 1
 Decimal:
  val=-4
  res=-8
 Binary:
  val=11111111111111111111111111111100
  res=11111111111111111111111111111000
 NOTE: zeros fill in right side

Expression: -2147483648 = -4 << 29
 Decimal:
  val=-4
  res=-2147483648
 Binary:
  val=11111111111111111111111111111100
  res=10000000000000000000000000000000

Expression: 0 = -4 << 30
 Decimal:
  val=-4
  res=0
 Binary:
  val=11111111111111111111111111111100
  res=00000000000000000000000000000000
 NOTE: bits shift out left side, including sign bit
```

以上例程在 64 位机器上的输出如下:

```
--- BIT SHIFT RIGHT ON POSITIVE INTEGERS ---
```

```
Expression: 2 = 4 >> 1
 Decimal:
  val=4
  res=2
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000010
 NOTE: copy of sign bit shifted into left side

Expression: 1 = 4 >> 2
 Decimal:
  val=4
  res=1
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000001

Expression: 0 = 4 >> 3
 Decimal:
  val=4
  res=0
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000000
 NOTE: bits shift out right side

Expression: 0 = 4 >> 4
 Decimal:
  val=4
  res=0
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000000000
 NOTE: same result as above; can not shift beyond 0


--- BIT SHIFT RIGHT ON NEGATIVE INTEGERS ---
Expression: -2 = -4 >> 1
 Decimal:
  val=-4
  res=-2
```

```
 Binary:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111110
 NOTE: copy of sign bit shifted into left side

Expression: -1 = -4 >> 2
 Decimal:
  val=-4
  res=-1
 Binary:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111111
 NOTE: bits shift out right side

Expression: -1 = -4 >> 3
 Decimal:
  val=-4
  res=-1
 Binary:
  val=11111111111111111111111111111111111111111111111111111111111111100
  res=11111111111111111111111111111111111111111111111111111111111111111
 NOTE: same result as above; can not shift beyond -1


--- BIT SHIFT LEFT ON POSITIVE INTEGERS ---
Expression: 8 = 4 << 1
 Decimal:
  val=4
  res=8
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=00000000000000000000000000000000000000000000000000000000000001000
 NOTE: zeros fill in right side

Expression: 4611686018427387904 = 4 << 60
 Decimal:
  val=4
  res=4611686018427387904
 Binary:
  val=00000000000000000000000000000000000000000000000000000000000000100
  res=01000000000000000000000000000000000000000000000000000000000000000
```

```
Expression: -9223372036854775808 = 4 << 61
 Decimal:
  val=4
  res=-9223372036854775808
 Binary:
  val=0000000000000000000000000000000000000000000000000000000000000100
  res=1000000000000000000000000000000000000000000000000000000000000000
 NOTE: sign bits get shifted out

Expression: 0 = 4 << 62
 Decimal:
  val=4
  res=0
 Binary:
  val=0000000000000000000000000000000000000000000000000000000000000100
  res=0000000000000000000000000000000000000000000000000000000000000000
 NOTE: bits shift out left side


--- BIT SHIFT LEFT ON NEGATIVE INTEGERS ---
Expression: -8 = -4 << 1
 Decimal:
  val=-4
  res=-8
 Binary:
  val=1111111111111111111111111111111111111111111111111111111111111100
  res=1111111111111111111111111111111111111111111111111111111111111000
 NOTE: zeros fill in right side

Expression: -9223372036854775808 = -4 << 61
 Decimal:
  val=-4
  res=-9223372036854775808
 Binary:
  val=1111111111111111111111111111111111111111111111111111111111111100
  res=1000000000000000000000000000000000000000000000000000000000000000

Expression: 0 = -4 << 62
 Decimal:
  val=-4
```

```
 res=0
Binary:
 val=11111111111111111111111111111111111111111111111111111111111111100
 res=00000000000000000000000000000000000000000000000000000000000000000
NOTE: bits shift out left side, including sign bit
```

在 32 位系统下向右移位时，不要超过 32 位，而且不要在结果可能超过 32 的情况下左移，可以使用 gmp 扩展对超出 PHP_INT_MAX 的数值来进行位操作。

22.3.3 Permission

在实践中的很多问题都可以简单地归结为 0 和 1 的问题，这样可以从新的角度来解决问题。例如，可以使用位运算来对 CMS 权限模型中的管理员、作者和访客等用户的不同权限进行分析。

具体来说，通过对二进制位的判断来实现对权限的分析

```php
<?php
define('PUBLIC_ARTICLE',1); // 发布文章 0000 0001
define('CREATE_ARTICLE',2); // 创建文章 0000 0010
define('MODIFY_ARTICLE',4); // 修改文章 0000 0100
define('DELETE_ARTICLE',8); // 删除文章 0000 1000
define('SEARCH_ARTICLE',16); // 搜索文章 0001 0000
define('CREATE_COMMENT',32); // 添加评论 0010 0000
define('DELETE_COMMENT',64); // 删除评论0100 0000

// 所有权限
$final_allow = PUBLIC_ARTICLE | CREATE_ARTICLE | MODIFY_ARTICLE | DELETE_ARTICLE |
    SEARCH_ARTICLE | CREATE_COMMENT | DELETE_COMMENT;
echo "管理者拥有的全部权限: " . decbin($final_allow) . "<br />";

$no_search_allow = $final_allow ^ SEARCH_ARTICLE;
echo "仅无法搜索文章的权限: " . decbin($no_search_allow) . "<br />";

$editor_allow = PUBLIC_ARTICLE | MODIFY_ARTICLE | DELETE_ARTICLE;
echo "编辑人员的权限: " . decbin($editor_allow) . "<br />";
echo "非编辑人员的权限: " . decbin($final_allow & ~$editor_allow) . "<br />";
?>
```

22.4 Comparison Operators

比较运算符允许对两个值进行比较。

表 22.5: PHP 比较运算符

运算符	说明	示例	结果
==	is equal to（等于）	$a == $b	TRUE，如果类型转换后 $a 等于 $b。
===	strict equal to（全等）	$a === $b	TRUE，如果 $a 等于 $b，并且它们的类型也相同。
!=	is not equal（不等）	$a != $b	TRUE，如果类型转换后 $a 不等于 $b。
<>	is not equal（不等）	$a <> $b	TRUE，如果类型转换后 $a 不等于 $b。
!==	not strict equal（不全等）	$a !== $b	TRUE，如果 $a 不等于 $b，或者它们的类型不同。
>	is greater than（大于）	$a > $b	TRUE，如果 $a 严格大于 $b。
<	is less than（小于）	$a < $b	TRUE，如果 $a 严格小于 $b。
>=	is greater than or equal to（大于等于）	$a >= $b	TRUE，如果 $a 大于或者等于 $b。
<=	is less than or equal to（小于等于）	$a <= $b	TRUE，如果 $a 小于或者等于 $b。
<=>		$a<=>$b	当 $a 小于、等于、大于 $b 时分别返回一个小于、等于、大于 0 的整数值
$a ?? $b ?? $c	NULL 合并操作符	从左往右第一个存在且不为 NULL 的操作数。如果都没有定义且不为 NULL，则返回 NULL。	

如果比较一个数字和字符串或者比较涉及到数字内容的字符串，则字符串会被转换为数值并且比较按照数值来进行，而且该规则也适用于 switch 语句。

当用 === 或!== 进行比较时则不进行类型转换，因为此时类型和数值都要比对，它们仅用于严格的比较场合。

- === 表示恒等，要求运算符两侧的表达式具有相同的类型和值；
- !== 表示非恒等，要求运算符两侧的表达式的类型和值都不同。

相比而言，== 和!= 只关心运算符两侧的表达式的值是否相等，并且在进行比较之前进行虑数据类型转换。

```
<?php
var_dump(0 == "a"); // 0 == 0 -> true
var_dump("1" == "01"); // 1 == 1 -> true
var_dump("10" == "1e1"); // 10 == 10 -> true
var_dump(100 == "1e2"); // 100 == 100 -> true

switch ("a") {
case 0:
    echo "0";
    break;
case "a": // never reached because "a" is already matched with 0
```

```
    echo "a";
    break;
}

// Integers
echo 1 <=> 1; // 0
echo 1 <=> 2; // -1
echo 2 <=> 1; // 1

// Floats
echo 1.5 <=> 1.5; // 0
echo 1.5 <=> 2.5; // -1
echo 2.5 <=> 1.5; // 1

// Strings
echo "a" <=> "a"; // 0
echo "a" <=> "b"; // -1
echo "b" <=> "a"; // 1

echo "a" <=> "aa"; // -1
echo "zz" <=> "aa"; // 1

// Arrays
echo [] <=> []; // 0
echo [1, 2, 3] <=> [1, 2, 3]; // 0
echo [1, 2, 3] <=> []; // 1
echo [1, 2, 3] <=> [1, 2, 1]; // 1
echo [1, 2, 3] <=> [1, 2, 4]; // -1

// Objects
$a = (object) ["a" => "b"];
$b = (object) ["a" => "b"];
echo $a <=> $b; // 0

$a = (object) ["a" => "b"];
$b = (object) ["a" => "c"];
echo $a <=> $b; // -1

$a = (object) ["a" => "c"];
$b = (object) ["a" => "b"];
echo $a <=> $b; // 1
```

```
// only values are compared
$a = (object) ["a" => "b"];
$b = (object) ["b" => "b"];
echo $a <=> $b; // 1
?>
```

对于多种类型，比较运算符根据下表比较（按顺序）。

表 22.6: PHP 比较多种类型

运算数 1 类型	运算数 2 类型	结果
null 或 string	string	将 NULL 转换为 ""，进行数字或词汇比较
bool 或 null	任何其它类型	转换为 bool，FALSE < TRUE
object	object	内置类可以定义自己的比较，不同类不能比较，相同类和数组同样方式比较属性（PHP 4）
string，resource 或 number	string，resource 或 number	将字符串和资源转换成数字，按普通数学比较
array	array	具有较少成员的数组较小，如果运算数 1 中的键不存在于运算数 2 中则数组无法比较，否则挨个值比较
object	任何其它类型	object 总是更大
array	任何其它类型	array 总是更大

由于浮点数 float 的内部表达方式，因此不应比较两个浮点数是否相等。

```
<?php
// 数组是用标准比较运算符进行比较
function standard_array_compare($op1, $op2)
{
    if (count($op1) < count($op2)) {
        return -1; // $op1 < $op2
    } elseif (count($op1) > count($op2)) {
        return 1; // $op1 > $op2
    }
    foreach ($op1 as $key => $val) {
        if (!array_key_exists($key, $op2)) {
            return null; // uncomparable
        } elseif ($val < $op2[$key]) {
            return -1;
        } elseif ($val > $op2[$key]) {
            return 1;
        }
    }
    return 0; // $op1 == $op2
}
```

```
?>
```

另一个条件运算符是“?:”（或三元）运算符。

```
<?php
 // Example usage for: Ternary Operator
 $action = (empty($_POST['action'])) ? 'default' : $_POST['action'];

 // The above is identical to this if/else statement
 if (empty($_POST['action'])) {
    $action = 'default';
 } else {
    $action = $_POST['action'];
 }

 ?>
```

表达式 (expr1) ? (expr2) : (expr3) 在 expr1 求值为 TRUE 时的值为 expr2，在 expr1 求值为 FALSE 时的值为 expr3。

三元运算符中间的部分可以省略。例如，表达式 expr1 ?: expr3 在 expr1 求值为 TRUE 时返回 expr1，否则返回 expr3。

需要注意的是，三元运算符本身是个语句，因此其求值不是变量，而是语句的结果。如果想通过引用返回一个变量这点就很重要。在一个通过引用返回的函数中语句 return $var == 42 ? $a : $b; 将不起作用，以后的 PHP 版本会为此发出一条警告。

应该避免将三元运算符堆积在一起使用。当在一条语句中使用多个三元运算符时会造成 PHP 运算结果不清晰。

```
<?php
// 乍看起来下面的输出是 'true'
echo (true?'true':false?'t':'f');

// 然而，上面语句的实际输出是't'，因为三元运算符是从左往右计算的

// 下面是与上面等价的语句，但更清晰
echo ((true ? 'true' : 'false') ? 't' : 'f');

// here, you can see that the first expression is evaluated to 'true', which
// in turn evaluates to (bool)true, thus returning the true branch of the
// second ternary expression.
?>
```

22.5 Error Control Operators

PHP 的错误控制运算符是 @，当将其放置在一个 PHP 表达式之前，该表达式可能产生的任何错误信息都被忽略掉。

如果用 set_error_handler() 设定了自定义的错误处理函数，仍然会被调用，但是此错误处理函数可以（并且也应该）调用 error_reporting()，而该函数在出错语句前有 @ 时将返回 0。

如果激活了 track_errors 特性，表达式所产生的任何错误信息都被存放在变量 $php_errormsg 中，该变量在每次出错时都会被覆盖，所以如果想用它的话就要尽早检查。

```
<?php
/* Intentional file error */
$my_file = @file ('non_existent_file') or
    die ("Failed opening file: error was '$php_errormsg'");

// this works for any expression, not just functions:
$value = @$cache[$key];
// will not issue a notice if the index $key doesn't exist.

?>
```

@ 运算符只对表达式有效，因此如果能从某处得到值，就能在它前面加上 @ 运算符。

- 可以把 @ 放在变量、函数和 include 调用、常量等等之前；
- 不能把 @ 放在函数或类的定义之前，也不能用于条件结构例如 if 和 foreach 等。

“@”错误控制运算符前缀可能使导致脚本终止的严重错误的错误报告也失效，因此如果在某个不存在或者敲错了字母的函数调用前用了“@”来抑制错误信息，那么脚本就会没有任何迹象显示原因就退出。

22.6 Execution Operators

PHP 的执行运算符是反引号（``），PHP 解析器将尝试将反引号中的内容作为 shell 命令来执行，并将其输出信息返回（即可以赋给一个变量而不是简单地丢弃到标准输出）。

反引号运算符“`”的效果与函数 shell_exec() 相同。

```
<?php
$output = `ls -al`;
echo "<pre>$output</pre>";
?>
```

反引号运算符在激活了安全模式或者关闭了 shell_exec() 时是无效的。

注意，与其它某些语言不同的是，反引号不能在双引号字符串中使用。

22.7 Incrementing/Decrementing

根据书写的方法不同，自增和自减运算符表达的意义也不同。

- 将表达式置于自增（或自减）运算符的前面，那么先返回表达式的值，然后再进行自增操作。
- 将表达式置于自增（或自减）运算符的前面，那么先进行自增（或自减）操作，然后再返回运算后的值。

表 22.7: PHP 算术运算符

运算符	说明	示例	结果
++$a	Increment	前加	$a 的值加一，然后返回 $a。
$a++	Increment	后加	返回 $a，然后将 $a 的值加一。
--$a	Decrement	前减	$a 的值减一，然后返回 $a。
$a--	Decrement	后减	返回 $a，然后将 $a 的值减一。

```
<?php
echo "<h3>Postincrement</h3>";
$a = 5;
echo "Should be 5: " . $a++ . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Preincrement</h3>";
$a = 5;
echo "Should be 6: " . ++$a . "<br />\n";
echo "Should be 6: " . $a . "<br />\n";

echo "<h3>Postdecrement</h3>";
$a = 5;
echo "Should be 5: " . $a-- . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";

echo "<h3>Predecrement</h3>";
$a = 5;
echo "Should be 4: " . --$a . "<br />\n";
echo "Should be 4: " . $a . "<br />\n";
?>
```

在处理字符变量的算数运算时，PHP 沿袭了 Perl 的习惯，而非 C 的。例如，在 Perl 中 `$a = 'Z'; $a++;` 将把 $a 变成'AA'，而在 C 中，`a = 'Z'; a++;` 将把 a 变成 '[' ('Z'

的 ASCII 值是 90，'[' 的 ASCII 值是 91)。

注意，字符变量只能递增，不能递减，并且只支持纯字母（a-z 和 A-Z)。递增/递减其他字符变量则无效，原字符串没有变化。

```
<?php
echo '== Alphabets ==' . PHP_EOL;
$s = 'W';
for ($n=0; $n<6; $n++) {
   echo ++$s . PHP_EOL;
}
// Digit characters behave differently
echo '== Digits ==' . PHP_EOL;
$d = 'A8';
for ($n=0; $n<6; $n++) {
   echo ++$d . PHP_EOL;
}
$d = 'A08';
for ($n=0; $n<6; $n++) {
   echo ++$d . PHP_EOL;
}
?>
```

以上例程会输出：

```
== Characters ==
X
Y
Z
AA
AB
AC
== Digits ==
A9
B0
B1
B2
B3
B4
A09
A10
```

```
A11
A12
A13
A14
```

- 递增/递减运算符不影响布尔值，递增或递减布尔值没有效果。
- 递减 NULL 值也没有效果，但是递增 NULL 的结果是 1。

22.8 Logical Operators

逻辑运算符“与”和“或”有两种不同形式运算符的原因是它们运算的优先级不同。

表 22.8: PHP 逻辑运算符

示例	运算符	结果
$a and $b	AND（逻辑与）	TRUE，如果 $a 和 $b 都为 TRUE。
$a or $b	OR（逻辑或）	TRUE，如果 $a 或 $b 任一为 TRUE。
$a xor $b	XOR（逻辑异或）	TRUE，如果 $a 或 $b 任一为 TRUE，但不同时是。
! $a	NOT（逻辑非）	TRUE，如果 $a 不为 TRUE。
$a&& $b	AND（逻辑与）	TRUE，如果 $a 和 $b 都为 TRUE。
$a\|\|$b	OR（逻辑或）	TRUE，如果 ab 任一为 TRUE。

```php
<?php

// --------------------
// foo() 根本没机会被调用，被运算符“短路”了

$a = (false && foo());
$b = (true || foo());
$c = (false and foo());
$d = (true or foo());

// --------------------
// "||" 比 "or" 的优先级高

// 表达式 (false || true) 的结果被赋给 $e
// 等同于：($e = (false || true))
$e = false || true;
```

```
// 常量 false 被赋给 $f, true 被忽略
// 等同于: (($f = false) or true)
$f = false or true;

var_dump($e, $f);

// --------------------
// "&&" 比 "and" 的优先级高

// 表达式 (true && false) 的结果被赋给 $g
// 等同于: ($g = (true && false))
$g = true && false;

// 常量 true 被赋给 $h, false 被忽略
// 等同于: (($h = true) and false)
$h = true and false;

var_dump($g, $h);
?>
```

以上例程的输出类似于:

```
bool(true)
bool(false)
bool(false)
bool(true)
```

PHP 的逻辑运算和 C 语言一样都支持“短路”计算，可以认为是对逻辑运算的优化，不过也可能导致短路部分的运算被忽略。例如，如果短路部分存在赋值运算，那么就会忽略它，因此将赋值语句放到逻辑判断中实际上是不合适的。

- 对于逻辑与运算，当从左边开始第一个表达式的值为假时就立即返回假，同时忽略后面的表达式求值。
- 对于逻辑或运算，当从左边开始第一个表达式的值为真时就立即返回真，同时忽略后面的表达式求值。

另外，“!”具有比“=”更高的优先级，因此 PHP 仍然支持传统的求反（或者说非）操作，不过这可能会产生误解。

```
<?php
if( !$a = rand(0,1) )
{
  echo $a;
```

```
}
?>
```

为了避免误解，上述的逻辑可以使用括号“()”来强调运算符的计算次序。

```
<?php
if( ($a=rand(0,1) == false )
{
  echo $a;
}
?>
```

22.9 String Operators

PHP 有两个字符串（string）运算符（. 和.=），而且它们都会将非字符串表达式自动转换为字符串类型。

- 连接运算符（“.”）返回其左右参数连接后的字符串。
- 连接赋值运算符（“.=”）将右边参数附加到左边的参数之后。

```
<?php
$a = "Hello ";
$b = $a . "World!"; // now $b contains "Hello World!"

$a = "Hello ";
$a .= "World!";  // now $a contains "Hello World!"
?>
```

22.10 Array Operators

表 22.9: PHP 数组运算符

示例	运算符	结果
$a + $b	联合	$a 和 $b 的联合。
$a == $b	相等	如果 $a 和 $b 具有相同的键/值对则为 TRUE。
$a === $b	全等	如果 $a 和 $b 具有相同的键/值对并且顺序和类型都相同则为 TRUE。
$a != $b	不等	如果 $a 不等于 $b 则为 TRUE。
$a <> $b	不等	如果 $a 不等于 $b 则为 TRUE。
$a !== $b	不全等	如果 $a 不全等于 $b 则为 TRUE。

+ 运算符把右边的数组元素附加到左边的数组后面，两个数组中都有的键名，则只用左边数组中的，右边的被忽略。

```
<?php
$a = array("a" => "apple", "b" => "banana");
$b = array("a" => "pear", "b" => "strawberry", "c" => "cherry");

$c = $a + $b; // Union of $a and $b
echo "Union of \$a and \$b: \n";
var_dump($c);

$c = $b + $a; // Union of $b and $a
echo "Union of \$b and \$a: \n";
var_dump($c);
?>
```

执行后，此脚本会显示：

```
Union of $a and $b:
array(3) {
  ["a"]=>
  string(5) "apple"
  ["b"]=>
  string(6) "banana"
  ["c"]=>
  string(6) "cherry"
}
Union of $b and $a:
array(3) {
  ["a"]=>
  string(4) "pear"
  ["b"]=>
  string(10) "strawberry"
  ["c"]=>
  string(6) "cherry"
}
```

数组中的单元如果具有相同的键名和值则比较时相等。

```
<?php
$a = array("apple", "banana");
```

```
$b = array(1 => "banana", "0" => "apple");

var_dump($a == $b); // bool(true)
var_dump($a === $b); // bool(false)
?>
```

22.11 Type Operators

instanceof 用于确定一个 PHP 变量是否属于某一类 class 的实例。

```
<?php
class MyClass
{
}

class NotMyClass
{
}
$a = new MyClass;

var_dump($a instanceof MyClass);
var_dump($a instanceof NotMyClass);
?>
```

以上例程会输出：

```
bool(true)
bool(false)
```

instanceof 也可用来确定一个变量是不是继承自某一父类的子类的实例。

```
<?php
class ParentClass
{
}

class MyClass extends ParentClass
{
}

$a = new MyClass;
```

```
var_dump($a instanceof MyClass);
var_dump($a instanceof ParentClass);
?>
```

以上例程会输出：

```
bool(true)
bool(true)
```

检查一个对象是否不是某个类的实例，可以使用逻辑运算符 not。

```
<?php
class MyClass
{
}

$a = new MyClass;
var_dump(!($a instanceof stdClass));
?>
```

以上例程会输出：

```
bool(true)
```

instanceof 也可用于确定一个变量是不是实现了某个接口的对象的实例。

```
<?php
interface MyInterface
{
}

class MyClass implements MyInterface
{
}

$a = new MyClass;

var_dump($a instanceof MyClass);
var_dump($a instanceof MyInterface);
?>
```

以上例程会输出：

```
bool(true)
bool(true)
```

虽然 instanceof 通常直接与类名一起使用，但是也可以使用对象或字符串变量。

```
<?php
interface MyInterface
{
}

class MyClass implements MyInterface
{
}

$a = new MyClass;
$b = new MyClass;
$c = 'MyClass';
$d = 'NotMyClass';

var_dump($a instanceof $b); // $b is an object of class MyClass
var_dump($a instanceof $c); // $c is a string 'MyClass'
var_dump($a instanceof $d); // $d is a string 'NotMyClass'
?>
```

以上例程会输出：

```
bool(true)
bool(true)
bool(false)
```

如果被检测的变量不是对象，instanceof 并不发出任何错误信息而是返回 FALSE。不允许用来检测常量。

```
<?php
$a = 1;
$b = NULL;
$c = imagecreate(5, 5);
var_dump($a instanceof stdClass); // $a is an integer
var_dump($b instanceof stdClass); // $b is NULL
var_dump($c instanceof stdClass); // $c is a resource
var_dump(FALSE instanceof stdClass);
?>
```

以上例程会输出：

```
bool(false)
```

```
bool(false)
bool(false)
PHP Fatal error:  instanceof expects an object instance, constant given
```

instanceof的使用的陷阱包括如果要检查的类名称不存在，instanceof会调用__autoload()（PHP 5.1.0之前）。

另外，如果该类没有被装载则会产生一个致命错误，可以通过使用动态类引用或用一个包含类名的字符串变量来避开这种问题。

```
<?php
$d = 'NotMyClass';
var_dump($a instanceof $d); // no fatal error here
?>
```

以上例程会输出：

```
bool(false)
```

instanceof运算符是PHP 5引进的，在此之前用is_a()，二者都可以使用。

22.11.1 is_a()

22.11.2 get_class()

Chapter 23

PHP Expression

表达式是 PHP 最重要的基石，通常的表达式是变量、常量和运算符的组合。

PHP 中几乎所写的任何东西都是一个表达式，因此简单但却最精确的定义一个表达式的方式就是“任何有值的东西”。

最基本的表达式形式是常量和变量。当键入“$a = 5”，即将值“5”分配给变量 $a。“5”，很明显其值为 5，换句话说“5”是一个值为 5 的表达式（在这里，“5”是一个整型常量)。

赋值之后，所期待情况是 $a 的值为 5，因而如果写下 $b = $a，期望的是它犹如 $b = 5 一样。换句话说，$a 也是一个值为 5 的表达式。如果一切运行正确，那这正是将要发生的正确结果。

稍微复杂的表达式例子就是函数。

```
<?php
function foo ()
{
   return 5;
}
?>
```

函数也是表达式，表达式的值即为它们的返回值。既然 foo() 返回 5，表达式“foo()”的值也是 5。

通常情况下，函数不会仅仅返回一个静态值，而可能会计算一些东西。

实际上，PHP 中的值常常并非是整型的，值可以是数字、字符串、数组、函数、对象以及其他表达式。例如，PHP 支持四种标量值（标量值不能拆分为更小的单元，例如和数组不同）类型：整型值（integer)，浮点数值（float)，字符串值（string）和布尔值（boolean)。PHP 也支持两种复合类型：数组和对象，这两种类型都可以赋值给变量或者从函数返回。

PHP 和其它语言一样在表达式的道路上发展，但是推进得更深远。PHP 是一种面向表达式的语言，从这一方面来讲几乎一切都是表达式。考虑刚才已经研究过的例子，“$a = 5”。

很显然这里涉及到两个值，整型常量“5”的值以及变量 $a 的值，它也被更新为 5。但是事实是这里还涉及到一个额外的值，即赋值语句本身的值。

赋值语句本身求值为被赋的值，即 5。实际上这意味着“$a = 5”，不必管它是做什么的，是一个值为 5 的表达式。因而，这样写“$b = ($a = 5)”和这样写“$a = 5; $b = 5”（分号标志着语句的结束）是一样的。因为赋值操作的顺序是由右到左的，也可以这么写“$b = $a = 5”。

另外一个很好的面向表达式的例子就是前、后递增和递减，PHP 和 C 语言一样都有两种类型的递增——前递增和后递增。本质上来讲，前递增和后递增均增加了变量的值，并且对于变量的影响是相同的，不同的是递增表达式的值。

- 前递增，写做“++$variable”，求增加后的值（PHP 在读取变量的值之前，增加变量的值，因而称之为“前递增”）。
- 后递增，写做“$variable++”，求变量未递增之前的原始值（PHP 在读取变量的值之后，增加变量的值，因而叫做“后递增”）。

下面的例子一般来说应该有助于理解前、后递增和表达式：

```
<?php
function double($i)
{
   return $i*2;
}
$b = $a = 5;      /* assign the value five into the variable $a and $b */
$c = $a++;        /* post-increment, assign original value of $a
                     (5) to $c */
$e = $d = ++$b;   /* pre-increment, assign the incremented value of
                     $b (6) to $d and $e */

/* at this point, both $d and $e are equal to 6 */

$f = double($d++); /* assign twice the value of $d before
                      the increment, 2*6 = 12 to $f */
$g = double(++$e); /* assign twice the value of $e after
                      the increment, 2*7 = 14 to $g */
$h = $g += 10;    /* first, $g is incremented by 10 and ends with the
                     value of 24. the value of the assignment (24) is
                     then assigned into $h, and $h ends with the value
                     of 24 as well. */
?>
```

一个常用到表达式类型是比较表达式。这些表达式求值为 FALSE 或 TRUE。PHP 支持 >（大于），>=（大于等于），==（等于），!=（不等于），<（小于）和 <=（小于等于）。

PHP 还支持全等运算符 ===（值和类型均相同）和非全等运算符!==（值或者类型不同）。

这些表达式都常用于条件判断语句中（比如 if 语句）。

这里，将要研究的最后一个例子是组合的运算赋值表达式。已经知道如果想要为变量 $a 加 1，可以简单的写“$a++”或者“++$a”。但是如果想为变量增加大于 1 的值，比如 3，该怎么做？可以多次写“$a++”，但这样明显不是一种高效舒适的方法，一个更加通用的做法是“$a = $a + 3”。“$a + 3”等值于 $a 加上 3 的值，并且得到的值重新赋予变量 $a，于是 $a 的值增加了 3。

在 PHP、C 和其他语言中，可以以一种更加简短的形式完成上述功能，因而也更加清楚快捷。为 $a 的当前值加 3，可以这样写：“$a += 3”。这里的意思是“取变量 $a 的值，加 3，得到的结果再次分配给变量 $a”。除了更简略和清楚外，也可以更快的运行。“$a += 3”的值，如同一个正常赋值操作的值，是赋值后的值。注意它不是 3，而是 $a 的值加上 3 之后的值（此值将被赋给 $a）。

任何二元运算符都可以用运算赋值模式，例如“$a -= 5”（从变量 $a 的值中减去 5），“$b *= 7”（变量 $b 乘以 7），等等。

PHP 同样支持三元条件运算符，例如：

```
<?php
$first ? $second : $third
?>
```

如果第一个子表达式的值是 TRUE（非零），那么第二个子表达式被求值，其值即为整个条件表达式的值。否则，第三个子表达式将被求值并且其值成为整个表达式的值。

一些表达式可以被当成语句。这时，一条语句的形式是 `expr;`，即一个表达式加一个分号结尾。在`'$b = $a = 5;'` 中，`'$a = 5'` 是一个有效的表达式，但是它本身不是一条语句，`'$b = $a = 5;'` 则是一条有效的语句。

最后一件值得提起的事情就是表达式的真值。在许多事件中，大体上主要是在条件执行和循环中，并不关心表达式中具体的值，而是只关心表达式的值是否是 TRUE 或者 FALSE。常量 TRUE 和 FALSE（大小写无关）是两种可能的布尔值。在必要时，一个表达式将自动转换为布尔值。

上面的例子应该提供了一个很好的关于什么是表达式和怎样构建一个有用的表达式的概念。

除了使用符号表达式之外，PHP 还支持正则表达式等字符串处理工具。

Chapter 24

PHP Array

在使用 PHP 进行开发的过程中，或早或晚，都会需要创建许多相似的变量，通过 PHP 数组就能够在单独的变量名中存储一个或多个值。

在 PHP 中，定义数组会用到 array 关键字，同时数组是可以定义索引的，方便快捷查询。

数组中的元素都有自己的 ID，因此可以方便地访问它们，PHP 有三种数组类型：

- 数值数组 - 带有数字 ID 键的数组
- 关联数组 - 数组中的每个 ID 键关联一个值
- 多维数组 - 包含一个或多个数组的数组

24.1 Numeric array

数值数组存储的每个元素都带有一个数字 ID 键，可以使用不同的方法来创建数值数组：

I 自动分配 ID 键

```
$names = array("Peter","Quagmire","Joe");
```

II 人工分配 ID 键

```
$names[0] = "Peter";
$names[1] = "Quagmire";
$names[2] = "Joe";
```

可以在脚本中使用这些 ID 键：

```
<?php

$names[0] = "Peter";
$names[1] = "Quagmire";
$names[2] = "Joe";
```

```
echo $names[1] . " and " . $names[2] . " are ". $names[0] . "'s neighbors";
?>
```

24.2 Associative array

关联数组，它的每个 ID 键都关联一个值。在存储有关具体命名的值的数据时，使用数值数组不是最好的做法。

通过关联数组，我们可以把值作为键，并向它们赋值。

在下面的示例中，使用一个数组把年龄分配给不同的人：

```
$ages = array("Peter"=>32, "Quagmire"=>30, "Joe"=>34);
```

本例与上面相同，不过展示了另一种创建数组的方法：

```
$ages['Peter'] = "32";
$ages['Quagmire'] = "30";
$ages['Joe'] = "34";
```

可以在脚本中使用 ID 键：

```
<?php

$ages['Peter'] = "32";
$ages['Quagmire'] = "30";
$ages['Joe'] = "34";

echo "Peter is " . $ages['Peter'] . " years old.";
?>
```

24.3 Multidimensional

在多维数组中，主数组中的每个元素也是一个数组。在子数组中的每个元素也可以是数组，以此类推。

下面的示例中创建了一个带有自动分配的 ID 键的多维数组：

```
$families = array
(
  "Griffin"=>array
  (
  "Peter",
  "Lois",
```

```
  "Megan"
  ),
  "Quagmire"=>array
  (
  "Glenn"
  ),
  "Brown"=>array
  (
  "Cleveland",
  "Loretta",
  "Junior"
  )
);
```

如果输出这个数组的话，应该类似这样：

```
Array
(
[Griffin] => Array
  (
  [0] => Peter
  [1] => Lois
  [2] => Megan
  )
[Quagmire] => Array
  (
  [0] => Glenn
  )
[Brown] => Array
  (
  [0] => Cleveland
  [1] => Loretta
  [2] => Junior
  )
)
```

如果要显示上面的数组中的一个单一的值：

```
echo "Is " . $families['Griffin'][2] . " a part of the Griffin family?";
```

24.4 Array Functions

PHP array 函数允许用户对数组进行操作，而且 PHP 支持单维和多维的数组，同时提供了用数据库查询结果来构造数组的函数。

PHP array 函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

表 24.1: PHP Array 函数

函数	描述	PHP
array()	创建数组。	3
array_change_key_case()	返回其键均为大写或小写的数组。	4
array_chunk()	把一个数组分割为新的数组块。	4
array_combine()	通过合并两个数组来创建一个新数组。	5
array_count_values()	用于统计数组中所有值出现的次数。	4
array_diff()	返回两个数组的差集数组。	4
array_diff_assoc()	比较键名和键值，并返回两个数组的差集数组。	4
array_diff_key()	比较键名，并返回两个数组的差集数组。	5
array_diff_uassoc()	通过用户提供的回调函数做索引检查来计算数组的差集。	5
array_diff_ukey()	用回调函数对键名比较计算数组的差集。	5
array_fill()	用给定的值填充数组。	4
array_filter()	用回调函数过滤数组中的元素。	4
array_flip()	交换数组中的键和值。	4
array_intersect()	计算数组的交集。	4
array_intersect_assoc()	比较键名和键值，并返回两个数组的交集数组。	4
array_intersect_key()	使用键名比较计算数组的交集。	5
array_intersect_uassoc()	带索引检查计算数组的交集，用回调函数比较索引。	5
array_intersect_ukey()	用回调函数比较键名来计算数组的交集。	5
array_key_exists()	检查给定的键名或索引是否存在于数组中。	4
array_keys()	返回数组中所有的键名。	4
array_map()	将回调函数作用到给定数组的单元上。	4
array_merge()	把一个或多个数组合并为一个数组。	4
array_merge_recursive()	递归地合并一个或多个数组。	4
array_multisort()	对多个数组或多维数组进行排序。	4

函数	描述	PHP
array_pad()	用值将数组填补到指定长度。	4
array_pop()	将数组最后一个单元弹出（出栈）。	4
array_product()	计算数组中所有值的乘积。	5
array_push()	将一个或多个单元（元素）压入数组的末尾（入栈）。	4
array_rand()	从数组中随机选出一个或多个元素，并返回。	4
array_reduce()	用回调函数迭代地将数组简化为单一的值。	4
array_reverse()	将原数组中的元素顺序翻转，创建新的数组并返回。	4
array_search()	在数组中搜索给定的值，如果成功则返回相应的键名。	4
array_shift()	删除数组中的第一个元素，并返回被删除元素的值。	4
array_slice()	在数组中根据条件取出一段值，并返回。	4
array_splice()	把数组中的一部分去掉并用其它值取代。	4
array_sum()	计算数组中所有值的和。	4
array_udiff()	用回调函数比较数据来计算数组的差集。	5
array_udiff_assoc()	带索引检查计算数组的差集，用回调函数比较数据。	5
array_udiff_uassoc()	带索引检查计算数组的差集，用回调函数比较数据和索引。	5
array_uintersect()	计算数组的交集，用回调函数比较数据。	5
array_uintersect_assoc()	带索引检查计算数组的交集，用回调函数比较数据。	5
array_uintersect_uassoc()	带索引检查计算数组的交集，用回调函数比较数据和索引。	5
array_unique()	删除数组中重复的值。	4
array_unshift()	在数组开头插入一个或多个元素。	4
array_values()	返回数组中所有的值。	4
array_walk()	对数组中的每个成员应用用户函数。	3
array_walk_recursive()	对数组中的每个成员递归地应用用户函数。	5
arsort()	对数组进行逆向排序并保持索引关系。	3
asort()	对数组进行排序并保持索引关系。	3
compact()	建立一个数组，包括变量名和它们的值。	4
count()	计算数组中的元素数目或对象中的属性个数。	3
current()	返回数组中的当前元素。	3
each()	返回数组中当前的键/值对并将数组指针向前移动一步。	3

函数	描述	PHP
end()	将数组的内部指针指向最后一个元素。	3
extract()	从数组中将变量导入到当前的符号表。	3
in_array()	检查数组中是否存在指定的值。	4
key()	从关联数组中取得键名。	3
krsort()	对数组按照键名逆向排序。	3
ksort()	对数组按照键名排序。	3
list()	把数组中的值赋给一些变量。	3
natcasesort()	用“自然排序”算法对数组进行不区分大小写字母的排序。	4
natsort()	用“自然排序”算法对数组排序。	4
next()	将数组中的内部指针向前移动一位。	3
pos()	current() 的别名。	3
prev()	将数组的内部指针倒回一位。	3
range()	建立一个包含指定范围的元素的数组。	3
reset()	将数组的内部指针指向第一个元素。	3
rsort()	对数组逆向排序。	3
shuffle()	把数组中的元素按随机顺序重新排列。	3
sizeof()	count() 的别名。	3
sort()	对数组排序。	3
uasort()	使用用户自定义的比较函数对数组中的值进行排序并保持索引关联。	3
uksort()	使用用户自定义的比较函数对数组中的键名进行排序。	3
usort()	使用用户自定义的比较函数对数组中的值进行排序。	3

24.5 Array Constants

表 24.2: PHP Array 常量

常量	描述	PHP
CASE_LOWER	用在 array_change_key_case() 中将数组键名转换成小写字母。	

常量	描述	PHP
CASE_UPPER	用在 array_change_key_case() 中将数组键名转换成大写字母。	
SORT_ASC	用在 array_multisort() 函数中，使其升序排列。	
SORT_DESC	用在 array_multisort() 函数中，使其降序排列。	
SORT_REGULAR	用于对对象进行通常比较。	
SORT_NUMERIC	用于对对象进行数值比较。	
SORT_STRING	用于对对象进行字符串比较。	
SORT_LOCALE_STRING	基于当前区域来对对象进行字符串比较。	4
COUNT_NORMAL		
COUNT_RECURSIVE		
EXTR_OVERWRITE		
EXTR_SKIP		
EXTR_PREFIX_SAME		
EXTR_PREFIX_ALL		
EXTR_PREFIX_INVALID		
EXTR_PREFIX_IF_EXISTS		
EXTR_IF_EXISTS		
EXTR_REFS		

Part V

Functions

Chapter 25

Introduction

PHP 提供了超过 700 个内建的函数，函数只需被编译一次就可以多次调用，因而 PHP 的真正威力源自于它的函数。

PHP 支持函数是“第一等公民”，即函数可以被赋值给一个变量（包括用户自定义的或者是内置函数），然后动态调用它。函数可以作为参数传递给其他函数（称为高阶函数），也可以作为函数返回值返回。

具体来说，函数是一种可以在任何被需要的时候执行的代码块，并且函数可以接受一组参数，然后返回操作结果。

PHP 有很多标准的函数和结构，还有一些函数需要和特定地 PHP 扩展模块一起编译，否则在使用它们的时候就会得到一个致命的“未定义函数”错误。

例如，要使用 image 函数中的 imagecreatetruecolor()，需要在编译 PHP 的时候加上 GD 的支持。或者，要使用 mysql_connect() 函数，就需要在编译 PHP 的时候加上 MySQL 支持。

PHP 的核心函数（例如字符串和变量函数）大多已经包含在每个版本的 PHP 中，调用 phpinfo() 或者 get_loaded_extensions() 可以得知 PHP 加载了哪些扩展库，而且很多扩展库默认就是有效的。

25.1 Function Prototype

函数的定义又可以称为函数原型，可以使用 function 声明自定义函数。

```
<?php
function foo($arg_1, $arg_2, /* ..., */ $arg_n)
{
    echo "Example function.\n";
    return $retval;
}
```

表 25.1: 函数原型

组成部分	说明
strlen	函数名称。
(PHP 4, PHP 5)	strlen() 在 PHP 4 和 PHP 5 的所有版本中都存在。
int	该函数返回的值的类型，这里为整型 integer（即以数字来衡量的字符串的长度）。
(string str)	第一个参数，在该函数中名为 str，且类型为 string。

```
?>
```

函数的原型说明了函数的返回值，或者函数是否直接作用于传递的参数。例如，str_replace() 函数将返回修改过的字符串，而 usort() 却直接作用于传递的参数变量本身。

- 所有的函数都使用关键词 “`function()`” 来开始
- 命名函数 - 函数的名称应该提示出它的功能，函数名称以字母或下划线开头。
- 添加 “`{`” - 开口的花括号之后的部分是函数的代码。
- 插入函数代码
- 添加一个 “`}`” - 函数通过关闭花括号来结束。

函数定义首先会告诉我们函数返回什么类型的值，以及函数有多少个变量。任何有效的 PHP 代码都有可能出现在函数内部，还可以包括其它函数和类定义。

函数名和 PHP 中的其它标识符命名规则相同，即有效的函数名应该以字母或下划线开始，后面跟字母、数字或下划线。

函数名是大小写无关的，不过在调用函数的时候，使用其在定义时相同的形式是个好习惯。

可以用正则表达式将函数名规则表示为：`[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*`。

下面用函数 strlen() 的定义作为第一个范例：

```
strlen
(PHP 4, PHP 5)
strlen -- 获取字符串长度
说明
int strlen ( string str )
返回给定的字符串 string 的长度。
```

可以将以上函数的定义写成一般形式：

```
返回类型          函数名          ( 参数类型          参数名 )
returned type    function name    ( parameter type   parameter name )
```

很多函数（例如 in_array()）都有多个变量，其函数原型如下：

```
bool in_array ( mixed $needle, array $haystack [, bool $strict])
```

in_array() 返回一个“布尔”值，成功（如果在参数 haystack 中能找到参数 needle）则返回 TRUE, 或者在失败时返回 FALSE（如果在参数 haystack 中找不到参数 needle）。第一个参数被命名为 needle 且其类型不定，因此我们将其称为“混和”类型。该混和类型的 needle 参数（我们要找的对象）可以是一个标量的值（字符串、整数、或者浮点数），或者一个数组。haystack（我们寻找的范围）是第二个参数。第三个可选参数被命名为 strict。所有的可选参数都用 [方括号] 括起来。

有的函数包含更复杂的 PHP 版本信息，例如：

```
(PHP 4 >= 4.3.0, PHP 5)
```

比较下面两个简单的函数，在其被调用时能输出名字。

```
<?php
function writeMyName() {
  echo "David Yang";
}

writeMyName();
?>
```

```
<?php
function writeMyName() {
  return "David Yang";
}

echo writeMyName();
?>
```

函数无需在调用之前被定义，除非函数是有条件被定义时。

/* 有条件的函数 */

```
<?php
$makefoo = true;
/* 不能在此处调用foo()函数,
   因为它还不存在，但可以调用bar()函数。*/
bar();
if ($makefoo) {
  function foo()
  {
    echo "I don't exist until program
        execution reaches me.\n";
  }
}
/* 现在可以安全调用函数 foo()了,
   因为 $makefoo 值为真 */
if ($makefoo) foo();
function bar()
{
  echo "I exist immediately upon program
       start.\n";
}
?>
```

/* 函数中的函数 */

```
<?php
function foo()
{
  function bar()
  {
    echo "I don't exist until foo() is
        called.\n";
  }
}

/* 现在还不能调用bar()函数，因为它还不存
    在 */

foo();

/* 现在可以调用bar()函数了，因为foo()函数
   的执行使得bar()函数变为已定义的函数 */

bar();

?>
```

当一个函数是有条件被定义时，其定义必须在调用之前先处理。

25.2 Function Iteration

PHP 中的函数可以在被调用之前定义，也可以在被调用之后定义，但是在某些情况下需要对函数的定义增加限制条件。

```
<?php
echo cal_circle_area(0.5);

function cal_circle_area($radius){
  return M_PI * ($radius * $radius);
}
?>
```

在函数的迭代中，低版本的 PHP 无法使用高版本 PHP 提供的更简便和实用的函数，同时还需要保证在不同版本中的兼容性，因此在使用 PHP 内置函数之前需要判断函数是否已

被定义。

```
<?php
/* 如果file_get_contents()函数不存在，则使用自定义的file_get_contents()函数版本 */
if(!function_exists("file_get_contents")){
  function file_get_contents($filename){
    $handle = fopen($filename,"r");
    $contents = fread($handle,filesize($filename));
    fclose($handle);
    return $contents;
  }
}

$string = file_get_contents("data.txt"); // 函数调用
?>
```

function_exists() 函数用于检查指定的函数是否存在，这样在不同版本的 PHP 环境之间切换时可以提高可移植性。

- 在低版本的 PHP 环境中使用自定义的函数；
- 在高版本的 PHP 环境中直接使用 PHP 内置的同名函数。

在上述的示例中，如果 PHP 版本小于 4.2.0 的环境中将会使用用户自己实现的 file_get_contents() 函数，反之则可以直接使用 PHP 内置的 file_get_contents() 函数。不过，为防止系统无法找到函数定义，需要在调用函数之前预先定义函数。

25.3 Function Scope

PHP 中的所有函数和类都具有全局作用域，可以在内部定义一个函数而在外部调用，反之亦然。

- PHP 不支持函数重载，也不可能取消定义或者重定义已声明的函数。
- PHP 的函数支持可变数量的参数和默认参数。

下面的示例演示了如何在 PHP 脚本中使用定义的函数：

```
<?php
function writeMyName() {
  echo "David Yang";
}

echo "Hello world!<br />";
echo "My name is ";
writeMyName();
echo ".<br />That's right, ";
```

```
writeMyName();
echo " is my name.";
?>
```

25.4 Recursive Functions

PHP 支持递归（也就是函数自己调用自己），虽然多数 PHP 代码使用迭代。

PHP 调用递归函数时要避免递归函数/方法调用超过 100-200 层，否则可能导致堆栈崩溃并使当前脚本终止。

```
<?php
function recursion($a)
{
    if ($a < 20) {
        echo "$a\n";
        recursion($a + 1);
    }
}
?>
```

递归本身是一种算法，即一个对象部分地包含自己，或者用自己给自己定义，例如 PHP 语言名称（PHP: Hypertext Preprocessor）本身就是递归的。

如果一个过程直接或间接地调用自己，那么就称其为递归的过程，例如数学上的阶乘函数、幂函数和斐波那契数列等的定义和计算都是递归的。

```
<?php
function Fractorial($n){
  if($n==0)
    return 1;
  else
    return n*Fractorial(n-1);
}
?>
```

递归函数必须要有终止递归的条件，否则函数将进入无限循环。

在使用 PHP 开发应用程序时，递归的应用包括文件目录的搜索和删除等，不过需要注意设计不合理的递归运算可能需要耗费大量的系统资源，因此在使用递归解决问题时需要限制递归的层数。

无限递归可视为编程错误。

25.5 Function Parameter

每个函数名称后面都有一个括号（比如 writeMyName()），参数就是在括号中规定的。例如，可以向函数传递数组：

```
<?php
function takes_array($input)
{
    echo "$input[0] + $input[1] = ", $input[0]+$input[1];
}
?>
```

下面的例子会输出不同的名字，但姓是相同的：

```
<?php
function writeMyName($fname)
  {
  echo $fname . " Yang.<br />";
  }

echo "My name is ";
writeMyName("David");

echo "My name is ";
writeMyName("Mike");

echo "My name is ";
writeMyName("John");
?>
```

如果传递给函数的参数类型与实际的类型不一致，例如将一个 array 传递给一个 string 类型的变量，那么函数的返回值是不确定的，虽然通常会返回 NULL，不过实际并不确定。

了解这些重要的（常常是细微的）差别是编写正确的 PHP 代码的关键。

- 添加参数可以向函数添加更多的功能，其中参数类似一个变量。
- 参数列表可以传递信息到函数，即以逗号作为分隔符的表达式列表，而且参数是从左向右求值的。
- 外部信息可以通过参数列表传入函数。

例如，下面示例中的函数的参数列表包含两个参数：

```
<?php
function writeMyName($fname,$punctuation)
  {
  echo $fname . " Yang" . $punctuation . "<br />";
```

```
 }

echo "My name is ";
writeMyName("David",".");

echo "My name is ";
writeMyName("Mike","!");

echo "My name is ";
writeMyName("John","...");
?>
```

- 在定义 PHP 函数时，带有默认值的参数必须放在参数列表的末尾。
- 在调用 PHP 函数时，将自动为没有赋值的参数赋予对应类型类型的默认值。

25.6 Parameter passing

PHP 支持按值传递参数，通过引用传递参数以及默认参数[1]，也支持可变长度参数列表。其中，可变长度参数并不需要特别的语法，参数列表仍按函数定义的方式传递给函数，并按通常的方式使用这些参数。

默认情况下，函数参数通过值传递，即使在函数内部改变参数的值，也不会改变函数外部的值。

- 如果希望函数内部的改变不影响到函数外部，可以按值来传递参数。
- 如果希望允许函数修改它的参数值，必须通过引用传递参数。

变量和变量地址类似于旅馆中的房间和房间号码，PHP 中的引用就是变量的“房间号码”，通过号码可以很容易找到变量。

引用可以高效地处理大型变量、数组和对象，总是使用引用向函数传递参数时可以在函数定义中的对应参数的前面加上符号 &，而且传引用的参数也可以有默认值。

```
<?php
function add_some_extra(&$string)
{
   $string .= 'and something extra.';
}
$str = 'This is a string, ';
add_some_extra($str);
echo $str;  // outputs 'This is a string, and something extra.'
?>
```

[1] 自 PHP 5 起，默认值才可以通过引用传递。

- C 语言变量的指针是符号表的别名，指针和变量本身存储在不同的内存地址中。
- PHP 变量的引用和变量具有相同的内容（内存地址）。

PHP 函数可以定义 C++ 风格的标量参数默认值。

```
<?php
function makecoffee($type = "cappuccino")
{
   return "Making a cup of $type.\n";
}
echo makecoffee();
echo makecoffee(null);
echo makecoffee("espresso");
?>
```

以上例程会输出：

```
Making a cup of cappuccino.
Making a cup of .
Making a cup of espresso.
```

PHP 还允许使用数组 array 和特殊类型 NULL 作为默认参数，例如：

```
<?php
function makecoffee($types = array("cappuccino"), $coffeeMaker = NULL)
{
   $device = is_null($coffeeMaker) ? "hands" : $coffeeMaker;
   return "Making a cup of ".join(", ", $types)." with $device.\n";
}
echo makecoffee();
echo makecoffee(array("cappuccino", "lavazza"), "teapot");
?>
```

默认值必须是常量表达式，不能是变量、类成员或者函数调用等。

注意，当使用默认参数时，任何默认参数必须放在任何非默认参数的右侧，否则函数将不会按照预期的情况工作。

```
<?php
function makeyogurt($type = "acidophilus", $flavour)
{
   return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry"); // won't work as expected
?>
```

以上例程会输出：

```
Warning: Missing argument 2 in call to makeyogurt() in
/usr/local/etc/httpd/htdocs/phptest/functest.html on line 41
Making a bowl of raspberry .
```

现在，比较上面的例子和这个例子：

```
<?php
function makeyogurt($flavour, $type = "acidophilus")
{
   return "Making a bowl of $type $flavour.\n";
}

echo makeyogurt("raspberry"); // works as expected
?>
```

以上例程会输出：

```
Making a bowl of acidophilus raspberry.
```

25.7 Type declarations

类型声明允许函数在被调用时接收确定类型的参数，传入类型不符的参数会导致错误。

- PHP5 产生可恢复的 FATAL error；
- PHP7 抛出 TypeError 错误。

如果需要指定类型声明，需要在参数列表中指定参数类型。如果参数的默认值为 NULL，那么参数声明可以接收 NULL。

表 25.2: 类型声明可接受的类型

类型	说明
class/interface name	指定的类或接口的名字
array	数组
callable	有效的回调
bool	布尔值
float	浮点数
int	整数
string	字符串

下面的示例说明了类型声明抛出 TypeError 错误的情况。

```
<?php
class C {}
class D extends C {}

// This doesn't extend C.
class E {}

function f(C $c) {
    echo get_class($c)."\n";
}

f(new C);
f(new D);
f(new E);
?>
```

上述示例的输出如下：

```
C
D

Fatal error: Uncaught TypeError: Argument 1 passed to f() must be an instance of C,
    instance of E given, called in - on line 14 and defined in -:8
Stack trace:
#0 -(14): f(Object(E))
#1 {main}
  thrown in - on line 8
```

如果类型声明中的参数要求是某个接口名，但是传入的参数未实现该接口也会抛出 TypeError 错误。

```
<?php
interface I { public function f(); }
class C implements I { public function f() {} }

// This doesn't implement I.
class E {}

function f(I $i) {
    echo get_class($i)."\n";
}
```

```
f(new C);
f(new E);
?>
```

上述示例会输出如下的结果：

```
C

Fatal error: Uncaught TypeError: Argument 1 passed to f() must implement interface I,
    instance of E given, called in - on line 13 and defined in -:8
Stack trace:
#0 -(13): f(Object(E))
#1 {main}
  thrown in - on line 8
```

如果类型声明中可以接受的类实例有默认值 NULL，那么就可以直接传入 NULL，例如：

```
<?php
class C {}

function f(C $c = null) {
    var_dump($c);
}

f(new C);
f(null);
?>
```

上述示例会输出：

```
object(C)#1 (0) {
}
NULL
```

25.8 Strict Typing

默认情况下，PHP 会强制将错误的类型转换成期望的标量类型，例如一个参数类型为 string 的函数来接受到整型变量时会将其转换为 string 类型。

如果开启强制类型，那么只有类型一致的变量才会被接受，否则会抛出 TypeError 错误。这里，唯一的例外是可以向类型声明为 float 的函数传入 int 型变量。

需要在 declare 语句中指定 strict_types 来启用严格模式，同时也会影响返回值的类型声明。

```
declare(strict_types=1);

function sum(int $a, int $b) {
    return $a + $b;
}

var_dump(sum(1, 2));
var_dump(sum(1.5, 2.5));
?>
```

如果不开启严格模式，那么上述示例是正常的，开启严格模式后就会输出：

```
int(3)

Fatal error: Uncaught TypeError: Argument 1 passed to sum() must be of the type integer,
     float given, called in - on line 9 and defined in -:4
Stack trace:
#0 -(9): sum(1.5, 2.5)
#1 {main}
  thrown in - on line 4
```

Strict typing applies to function calls made from within the file with strict typing enabled, not to the functions declared within that file. If a file without strict typing enabled makes a call to a function that was defined in a file with strict typing, the caller's preference (weak typing) will be respected, and the value will be coerced.

Strict typing is only defined for scalar type declarations, and as such, requires PHP 7.0.0 or later, as scalar type declarations were added in that version.

如果需要捕获 TypeError 错误，可以使用 try/catch，例如：

```
<?php
declare(strict_types=1);

function sum(int $a, int $b) {
    return $a + $b;
}

try {
    var_dump(sum(1, 2));
    var_dump(sum(1.5, 2.5));
} catch (TypeError $e) {
    echo 'Error: '.$e->getMessage();
}
?>
```

以上例程会输出：

```
int(3)
Error: Argument 1 passed to sum() must be of the type integer, float given, called in -
    on line 10
```

25.9 Variadic arguments

PHP 在用户自定义函数中支持可变数量的参数列表。

在 PHP 5.5 及更早版本中使用 func_num_args()，func_get_arg() 和 func_get_args() 函数，在 PHP 5.6 及以上的版本中可以由 ... 语法实现。

- func_get_args() 函数作用于自定义函数内部，返回一个包含所有传递给函数的参数的数组。

```
<?php
function more_args(){
  $args = func_get_args();
  foreach($args as $current_arg){
    echo $current_arg . PATH_SEPARATOR;
  }
}
more_args("A","B","C");
?>
```

- func_num_args() 函数返回参数的总数。

```
<?php
function more_args(){
  $num = func_num_args();
  for($i=0; $i<$num; $i++){
    $current_arg = func_get_arg($i);
    echo $current_arg . PATH_SEPARATOR;
  }
}
?>
```

- func_num_arg() 函数接受一个数字参数，并返回指定的参数。

在 PHP5.6 以及以后，参数列表可以接受... 来说明函数可以接受可变数量的参数列表，而且支持以数组的形式向函数传入变量。例如，下面的示例就可以把接收到的变量列表以数组的形式进行存储和处理，最终输出结果为 10。

```
<?php
function sum(...$numbers) {
```

```
    $acc = 0;
    foreach ($numbers as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

支持可变数量的参数列表的函数也可以用来解包数组或 Traversable 类型的变量或者遍历参数列表，例如：

```
<?php
function add($a, $b) {
    return $a + $b;
}

echo add(...[1, 2])."\n";

$a = [1, 2];
echo add(...$a);
?>
```

以上例程会输出：

```
3
3
```

You may specify normal positional arguments before the ... token. In this case, only the trailing arguments that don't match a positional argument will be added to the array generated by ....

It is also possible to add a type hint before the ... token. If this is present, then all arguments captured by ... must be objects of the hinted class.

```
<?php
function total_intervals($unit, DateInterval ...$intervals) {
    $time = 0;
    foreach ($intervals as $interval) {
        $time += $interval->$unit;
    }
    return $time;
}

$a = new DateInterval('P1D');
```

```
$b = new DateInterval('P2D');
echo total_intervals('d', $a, $b).' days';

// This will fail, since null isn't a DateInterval object.
echo total_intervals('d', null);
?>
```

以上例程会输出：

```
3 days
Catchable fatal error: Argument 2 passed to total_intervals() must be an instance of
    DateInterval, null given, called in - on line 14 and defined in - on line 2
```

You may also pass variable arguments by reference by prefixing the ... with an ampersand (&).

在旧版本的 PHP 中没有指定的语法来说明函数是否接收可变数量的参数列表，需要 func_num_args()、func_get_arg() 和 func_get_args()。

```
<?php
function sum() {
    $acc = 0;
    foreach (func_get_args() as $n) {
        $acc += $n;
    }
    return $acc;
}

echo sum(1, 2, 3, 4);
?>
```

上述示例输出的结果同样会是 10。

25.9.1 func_get_args()

25.9.2 func_num_args()

25.9.3 func_num_arg()

25.10 Return Values

函数中的显式的 return 语句表示函数执行结束，并返回执行结果。如果省略了 return，则返回值为 NULL，这样函数的作用和过程类似。

函数也能用于返回值，值通过使用可选的返回语句返回，可以返回的值包括数组和对象的任意类型，而且返回语句会立即中止函数的运行，并且将控制权交回调用该函数的代码行。

实际上，函数的参数和返回值类型都是不受限制的，可以是标量，也可以是对象或资源，而且参数列表和 return 都不是必须的。

```
<?php
function add($x,$y)
  {
  $total = $x + $y;
  return $total;
  }

echo "1 + 16 = " . add(1,16);
?>
```

函数不能返回多个值，但是可以通过返回一个数组来得到类似的效果。

```
<?php
function small_numbers()
{
   return array (0, 1, 2);
}
list ($zero, $one, $two) = small_numbers();
?>
```

从函数返回一个引用，必须在函数声明和指派返回值给一个变量时都使用引用运算符 &。

```
<?php
function &returns_reference()
{
   return $someref;
}

$newref =& returns_reference();
?>
```

从 PHP7 开始支持返回值类型声明，而且严格类型模式同时影响参数类型声明和返回值类型声明，类型错误都会抛出 TypeError 错误。

如果需要重写父类的方法，子类的方法必须可以匹配父类中的任何返回值类型声明，即使父类中没有指定返回类型。

```
<?php
function sum($a, $b): float {
   return $a + $b;
}
```

```
// Note that a float will be returned.
var_dump(sum(1, 2));
?>
```

以上例程会输出：

```
float(3)
```

下面的示例中进行严格类型声明：

```
<?php
declare(strict_types=1);

function sum($a, $b): int {
    return $a + $b;
}

var_dump(sum(1, 2));
var_dump(sum(1, 2.5));
?>
```

以上例程会输出：

```
int(3)

Fatal error: Uncaught TypeError: Return value of sum() must be of the type integer,
    float returned in - on line 5 in -:5
Stack trace:
#0 -(9): sum(1, 2.5)
#1 {main}
  thrown in - on line 5
```

如果在函数定义时指定了返回值类型，那么必须返回值必须匹配声明。

```
<?php
class C {}

function getC(): C {
    return new C;
}

var_dump(getC());
?>
```

以上例程会输出：

```
object(C)#1 (0) {
}
```

Chapter 26

Variable functions

PHP 支持可变函数的概念，这样如果一个变量名后有圆括号，PHP 将寻找与变量的值同名的函数，并且尝试执行它。

可变函数可以用来实现包括回调函数和函数表等，但是可变函数不能用于例如 echo，print，unset()，isset()，empty()，include，require 以及类似的语言结构，需要使用自己的包装函数来将这些结构用作可变函数。

```
<?php
function foo() {
    echo "In foo()<br />\n";
}

function bar($arg = '') {
    echo "In bar(); argument was '$arg'.<br />\n";
}

// 使用 echo 的包装函数
function echoit($string)
{
    echo $string;
}

$func = 'foo';
$func();      // This calls foo()

$func = 'bar';
$func('test'); // This calls bar()
```

```
$func = 'echoit';
$func('test'); // This calls echoit()
?>
```

也可以用可变函数的语法来调用一个对象的方法。

```
<?php
class Foo
{
    function Variable()
    {
        $name = 'Bar';
        $this->$name(); // This calls the Bar() method
    }

    function Bar()
    {
        echo "This is Bar";
    }
}

$foo = new Foo();
$funcname = "Variable";
$foo->$funcname(); // This calls $foo->Variable()

?>
```

当调用静态方法时，函数调用要比静态属性优先：

```
<?php
class Foo
{
    static $variable = 'static property';
    static function Variable()
    {
        echo 'Method Variable called';
    }
}

echo Foo::$variable; // This prints 'static property'. It does need a $variable in this
    scope.
$variable = "Variable";
Foo::$variable(); // This calls $foo->Variable() reading $variable in this scope.
?>
```

Chapter 27

Anonymous functions

匿名函数（Anonymous functions）也叫闭包函数（closures），允许临时创建一个没有指定名称的函数。

匿名函数经常用作回调函数（callback）参数的值，当然也有其它应用的情况。

```
<?php
echo preg_replace_callback('~-([a-z])~', function ($match) {
    return strtoupper($match[1]);
}, 'hello-world');
// 输出 helloWorld
?>
```

闭包函数也可以作为变量的值来使用，而且匿名函数通过 Closure 类来实现的，因此 PHP 会自动把此种表达式转换成内置类 Closure 的对象实例。

把一个 closure 对象赋值给一个变量的方式与普通变量赋值的语法是一样的，最后也要加上分号。

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

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

闭包可以从父作用域中继承变量，任何此类变量都应该用 use 语言结构传递进去，这些变量都必须在函数或类的头部声明。

Example 28 从父作用域继承变量

```
<?php
$message = 'hello';

// 没有 "use"
$example = function () {
    var_dump($message);
};
echo $example();

// 继承 $message
$example = function () use ($message) {
    var_dump($message);
};
echo $example();

// Inherited variable's value is from when the function
// is defined, not when called
$message = 'world';
echo $example();

// Reset message
$message = 'hello';

// Inherit by-reference
$example = function () use (&$message) {
    var_dump($message);
};
echo $example();

// The changed value in the parent scope
// is reflected inside the function call
$message = 'world';
echo $example();

// Closures can also accept regular arguments
$example = function ($arg) use ($message) {
    var_dump($arg . ' ' . $message);
};
$example("hello");
?>
```

以上例程的输出类似于：

```
Notice: Undefined variable: message in /example.php on line 6
NULL
string(5) "hello"
string(5) "hello"
string(5) "hello"
string(5) "world"
string(11) "hello world"
```

从父作用域中继承变量与使用全局变量是不同的，其中：

- 全局变量存在于一个全局的范围，无论当前在执行的是哪个函数。
- closure 的父作用域则是声明该 closure 的函数（不一定要是它被调用的函数）。

```
<?php
// 一个基本的购物车，包括一些已经添加的商品和每种商品的数量。
// 其中有一个方法用来计算购物车中所有商品的总价格，该方法使
// 用了一个 closure 作为回调函数。
class Cart
{
    const PRICE_BUTTER = 1.00;
    const PRICE_MILK = 3.00;
    const PRICE_EGGS = 6.95;

    protected $products = array();

    public function add($product, $quantity)
    {
        $this->products[$product] = $quantity;
    }

    public function getQuantity($product)
    {
        return isset($this->products[$product]) ? $this->products[$product] :
              FALSE;
    }

    public function getTotal($tax)
    {
        $total = 0.00;

        $callback =
            function ($quantity, $product) use ($tax, &$total)
```

```
            {
                $pricePerItem = constant(__CLASS__ . "::PRICE_" .
                    strtoupper($product));
                $total += ($pricePerItem * $quantity) * ($tax + 1.0);
            };

        array_walk($this->products, $callback);
        return round($total, 2);;
    }
}

$my_cart = new Cart;

// 往购物车里添加条目
$my_cart->add('butter', 1);
$my_cart->add('milk', 3);
$my_cart->add('eggs', 6);

// 打出出总价格，其中有 5% 的销售税.
print $my_cart->getTotal(0.05) . "\n";
// 最后结果是 54.29
?>
```

和可变长度的参数列表的函数相同，可以在 closure 中使用 func_num_args()，func_get_arg() 和 func_get_args()，而且 $this 可用于匿名函数。

Chapter 28

PHP Script

现在来编写一些更实用的脚本，比如检查浏览页面的访问者在用什么浏览器。要达到这个目的，需要检查用户的 agent 字符串，它是浏览器发送的 HTTP 请求的一部分。该信息被存储在一个变量中。在 PHP 中，变量总是以 $ 开头。

现在要用到的变量是 $_SERVER['HTTP_USER_AGENT']，而 $_SERVER 是一个特殊的 PHP 保留变量，它包含了 Web 服务器提供的所有信息，被称为**超全局变量**。这些特殊的变量是在 PHP 4.1.0 版本引入的，在这之前使用 $HTTP_*_VARS 数组，如 $HTTP_SERVER_VARS。尽管现在已经不用了，但它们在新版本中仍然存在。

要显示 $_SERVER['HTTP_USER_AGENT'] 变量，只需简单地进行如下操作：

```
<?php
echo $_SERVER['HTTP_USER_AGENT'];
?>
```

PHP 有很多种不同类型的变量，$_SERVER 只是 PHP 自动全局化的变量之一。在以上例子中我们打印了一个数组的单元，而数组是一类非常有用的变量。

可以在一个 PHP 标识中加入多个 PHP 语句，也可以建立一个代码块来做比简单的 echo 更多的事情。例如，如果需要识别 Internet Explorer，可以进行如下操作：

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'], 'MSIE') !== FALSE) {
   echo '正在使用 Internet Explorer。<br />';
}
else{
   echo '你使用的不是Internet Explorer。 ';
}
?>
```

其中，strpos() 是 PHP 的一个内置函数，其功能是在一个字符串中搜索另外一个字符串。例如我们现在需要在 $_SERVER['HTTP_USER_AGENT']（即所谓的 haystack）变量中寻找'MSIE'。如果在这个 haystack 中该字符串（即所谓的 needle）被找到（“草里寻针”），则函数返回 needle 在 haystack 中相对于开头的位置；如果没有，则返回 FALSE。如果该函数没有返回 FALSE，则 if 会将条件判断为 TRUE 并运行其花括号 {} 内的代码；否则，则不运行这些代码。可以自己尝试利用 if，else 以及其它的函数如 strtoupper() 和 strlen() 来建立类似的脚本。

以下我们进一步显示如何进出 PHP 模式，甚至是在一个 PHP 代码块的中间：

```
<?php
if (strpos($_SERVER['HTTP_USER_AGENT'], 'MSIE') !== FALSE) {
?>
<h3>strpos() 肯定没有返回假 (FALSE)</h3>
<p>正在使用 Internet Explorer</p>
<?php
} else {
?>
<h3>strpos() 肯定返回假 (FALSE)</h3>
<center><b>没有使用 Internet Explorer</b></center>
<?php
}
?>
```

和以上我们用一个 PHP 的 echo 语句来输出不同的是，我们跳出了 PHP 模式来直接写 HTML 代码。这里很值得注意的一点是，对于这两种情况而言，脚本的逻辑效率是相同的。在判断了 strpos() 函数的返回值是 TRUE 或是 FALSE，也就是判断了字符串'MSIE' 是否被找到之后，最终只有一个 HTML 块被发送给浏览者。

Chapter 29

PHP Library Functions

Chapter 30

PHP Array

30.1 Key

30.1.1 array_values()

array_values() 函数可以返回数组中的所有值，而且忽略原始的键名，并使用顺序的数字对数组重新索引。

```
array array_values ( array $input )
```

30.1.2 array_keys()

array_keys() 函数可以返回一个数组中的所有键，其返回值是一个包含数字或字符串的键名数组。

```
array array_keys ( array $array [, mixed $search_value [, bool $strict = false ]] )
```

array_values() 和 array_keys() 函数都把数组作为一维的进行处理，并且保持元素值或键名的原始顺序。

30.1.3 array_flip()

array_flip() 返回一个反转后的 array，例如 trans 中的键名变成了值，而 trans 中的值成了键名。

```
array array_flip ( array $trans )
```

值和键名的交换是有条件的，数组的值可以重复将导致交换后的数组中发生覆盖。

30.1.4 in_array()

```
bool in_array ( mixed $needle , array $haystack [, bool $strict = FALSE ] )
```

30.1.5 array_search()

array_search() 函数和 in_array() 函数的参数相同，而且都支持对数据类型的严格判断。

```
mixed array_search ( mixed $needle , array $haystack [, bool $strict = false ] )
```

30.1.6 array_key_exists()

array_key_exists() 函数可以检索给定的键名（索引）是否存在于数组中。

```
bool array_key_exists ( mixed $key , array $search )
```

数组的键名是唯一的，也无需对其数据类型进行判断，因此在检索给定的键名时使用 isset() 函数是一个更好的选择。

```
if(array_key_exists("num",$var_array)){
  echo $var_array[$key];
}

// isset()是更为常见的解决方案
if(isset($var_array["num"])){
  echo $var_array[$key];
}
```

30.2 Pointer

指针是 PHP 数组的内部组织机制，内部指针指向一个数组单元，或者说对应着一个数组元素。

- 在数组初始化时，或者将一个数组赋值给另一个数组时，指针将指向数组最开始的元素；
- 在数组初始化后，通过移动或改变指针位置，可以访问数组的任意元素。

30.2.1 current()

```
mixed current ( array &$array )
```

30.2.2 key()

```
mixed key ( array &$array )
```

30.2.3 prev()

```
mixed prev ( array &$array )
```

30.2.4 next()

```
mixed next ( array &$array )
```

30.2.5 end()

```
mixed end ( array &$array )
```

30.2.6 reset()

```
mixed reset ( array &$array )
```

30.2.7 each()

each() 函数返回数组当前元素的一个键名/值的构造数组，并使数组指针向前移动一位。

```
array each ( array &$array )
```

30.2.8 list()

each() 和 list() 可以配合使用来进行数组的遍历操作，不过 list() 仅能用于数字索引的数组，并假定数字索引从 0 开始。

用户可以使用下面的表达式来直接获得数组中的当前元素的键名和值。

```
list($index, $fruit) = each($arr)
```

30.3 Variable

extract() 和 compact() 函数可以进行数组和变量之间的转换，并且转换后的数组元素的键名和变量名，以及元素的值和元素的值保持着对应的关系。

30.3.1 extract()

```
int extract ( array &$var_array [, int $extract_type = EXTR_OVERWRITE [, string $prefix
    = NULL ]] )
```

30.3.2 compact()

```
array compact ( mixed $varname [, mixed $... ] )
```

30.4 Segment

对于较大的数组，可以进行分段来提高操作效率。

30.4.1 array_slice()

```
array array_slice ( array $array , int $offset [, int $length = NULL [, bool
    $preserve_keys = false ]] )
```

30.4.2 array_splice()

30.4.3 array_chunk()

```
array array_chunk ( array $input , int $size [, bool $preserve_keys = false ] )
```

30.5 Padding

30.5.1 array_pad()

```
array array_pad ( array $input , int $pad_size , mixed $pad_value )
```

30.6 Stack

PHP 数组也可以作为栈（数组栈）使用，栈底指向数组的第一个元素，栈顶指向数组中的最后一个元素。

对数组栈的操作包括入栈和出栈，而且 PHP 提供了 array_push() 和 array_pop() 函数来实现数组栈元素的压入和弹出。

30.6.1 array_push()

30.6.2 array_pop()

30.7 Queue

PHP 数组可以实现对队列的模拟，这样就可以在队列的一端插入数组，并且在另一侧删除数据，这样最先进入队列的数据将最先离开队列。

- 允许插入的一端称为队尾，即数组的第一个元素；
- 允许删除的一端称为队头，即数组的最后一个元素。

30.7.1 array_shift()

30.7.2 array_unshift()

30.8 Callback

回调机制允许用户使用自定义的方法对数据进行操作，例如回调函数可以用于对数组的处理。

30.8.1 array_walk()

array_walk() 函数可以使用用户自定义的函数来对数组中的每个成员进行处理，因此也称为单一数组回调处理函数。

```
bool array_walk ( array &$array , callable $funcname [, mixed $userdata = NULL ] )
```

在 array_walk() 函数中使用回调函数可以直接改变数组的元素值，但是对于键名的更改是无效的。

30.8.2 array_map()

```
array array_map ( callable $callback , array $arr1 [, array $... ] )
```

回调函数平行作用于相应的数组元素上，因此 array_map() 函数用于两个或多个数组时，它们的长度应该相同，否则最短的数组将用 NULL 元素进行扩充。

和使用单个数组不同，array_map() 函数忽略多个数组中的键名，并统一分配数字索引作为键名。

30.8.3 array_filter()

```
array array_filter ( array $input [, callable $callback = "" ] )
```

30.8.4 array_reduce()

array_reduce() 将回调函数 function 迭代地作用到 input 数组中的每一个单元中，从而将数组简化为单一的值。

```
mixed array_reduce ( array $input , callable $function [, mixed $initial = NULL ] )
```

30.9 Sort

PHP 可以通过元素或索引进行排序，也可以使用自然排序法或用户自定义的排序方法等进行排序。

30.9.1 sort()

30.9.2 rsort()

30.9.3 usort()

30.9.4 asort()

30.9.5 arsort()

30.9.6 uasort()

30.9.7 ksort()

30.9.8 krsort()

30.9.9 uksort()

30.9.10 natsort()

natsort() 函数可以使用自然排序法对包含文件名的数组进行排序，而且排序结构忽略键名。

natsort() 函数对大小写不敏感，如果需要使用键名和值对应的“自然排序”，可以使用 uasort() 和 strnatcmp() 函数的替代方式。

30.10 Computation

通常情况下，对数组的计算包括求数组内部的所有元素之和，或者把数组作为一个集合进行处理，以及对两个或多个数组进行合并，计算数组之间的差集或交集等。

30.10.1 array_sum()

30.10.2 array_merge()

30.10.3 array_merge_recursive()

30.10.4 array_diff()

30.10.5 array_intersect()

和计算数组的差集类似，数组的交集仅计算多维数组中的一维，因此需要使用索引进行依次计算才能正确地处理多维数组。

在交集运算的结果数组中，键名将保持不变。

30.10.6 array_intersect_assoc()

30.11 Unique

30.11.1 array_unique()

array_unique() 函数可以移除数组中重复的值，并且返回以恶没有重复元素的新数组。

一般情况下，数组中的重复元素指元素的字符串表达式相同，新的数组中将会保留元素原始的键名。

30.12 Reverse

30.12.1 array_reverse()

30.13 Random

30.13.1 array_rand()

array_rand() 从数组中随机取出一个或多个单元，并返回随机条目的一个或多个键。

```
mixed array_rand ( array $input [, int $num_req = 1 ] )
```

在下面的示例中使用随机数发生器种子来随机输出字符串。

```
<?php

// 设置随机数发生器种子
srand((float)microtime() * 10000000);

$servers = array("UNIX","Linux","Windows","Mac OS X","Solaris");
$rand_keys = array_rand($servers,2);
```

```
print $servers[$rand_keys[0]]."\n";
print $servers[$rand_keys[1]]."\n";
?>
```

在处理随机问题时，通常必须首先设置随机发生器的种子，这样才能让随机数功能正常执行。

30.13.2 shuffle()

shuffle() 函数可以将数组的顺序打乱，其实质也是一个随机化过程，因此需要设置随机数发生器种子。

```
<?php
$numbers = range(1,20);
srand((float)microtimes() * 1000000);
shuffle($numbers);

// 循环输出
while(list(,$number) = each ($numbers))
{
  echo "$number = ";
}
?>
```

30.14 Range

30.14.1 range()

```
array range ( mixed $start , mixed $limit [, number $step = 1 ] )
```

Chapter 31

PHP Date/Time

31.1 Timestamp

时间戳（timestamp）是指在一连串的信息中加入辨识文字（例如时间或日期），用以保障本地端（local）信息更新顺序与远端（remote）一致。

时间戳的范例如下：

```
2005-10-30 T 10:45 UTC
2007-11-09 T 11:20 UTC
Sat Jul 23 02:16:57 2005
```

UNIX 时间[1]（或称 UNIX 时间戳，POSIX 时间）是一种在 UNIX 或类 UNIX 系统中使用的时间表示方式，其纪元时间从 UTC（协调世界时）1970 年 1 月 1 日 0 时 0 分 0 秒开始，因此 UNIX 时间戳就是从 UNIX TIME 纪元时间起至现在的总秒数（不考虑闰秒）。

UNIX 时间戳是一个有符号整数，所以 1970 年前一百年可以用其负数部分表示，但是事实上很少这样使用。因为 UNIX 时间戳主要用来表示当前时间或者和计算机有关的日志时间 (如文件创建时间和 log 发生时间等)。

考虑到所有计算机文件不可能在 1970 年前创建，所以用 UNIX 时间戳很少用来表示 1970 前的时间。如果需要表示以前的时间，一般就是自己定义数据结构，例如可以用几个数分别表示年月日，或者类似 Excel 中用 1900 年 1 月 1 日后的天数表示时间。

后来的计算机时间、UNIX 时间、Linux 时间、Java 时间都是以 1970 年 1 月 1 日 0 时 0 分 0 秒为原点开始计算。例如，Integer 在 Java 内用 32 位表示，因此 32 位能表示的最大值是 2147483647。

```
System.out.println(Integer.MAX_VALUE);
```

[1]UNIX 时间以 5000 日为纪念日，第一个 5000 日是 1983 年 9 月 10 日，第 10000 日是 1997 年 5 月 19 日，第 15000 日是 2011 年 1 月 26 日，第 20000 日是 2024 年 10 月 4 日。

```
2147483647
```

表 31.1: 在不同编程语言中获取现在的 UNIX 时间戳

语言	操作
Java	`time`
JavaScript	`Math.round(new Date().getTime()/1000)`，getTime() 返回数值的单位是毫秒
Microsoft .NET / C#	`epoch = (DateTime.Now.ToUniversalTime().Ticks - 621355968000000000) / 10000000`
MySQL	`SELECT unix_timestamp(now())`
Perl	`time`
PHP	`time()`
PostgreSQL	`SELECT extract(epoch FROM now())`
Python	先 `import time` 然后 `time.time()`
Ruby	获取 Unix 时间戳：`Time.now` 或 `Time.new` 显示 Unix 时间戳：`Time.now.to_i`
SQL Server	`SELECT DATEDIFF(s, '1970-01-01 00:00:00', GETUTCDATE())`
Unix / Linux	`date +%s`
VBScript / ASP	`DateDiff("s", "01/01/1970 00:00:00", Now())`
Perl 命令行状态	`perl -e "print time"`

另外，1 年 365 天的总秒数是 31536000，2147483647/31536000 = 68.1，也就是说 32 位能表示的最长时间是 68 年，而实际上到 2038 年 01 月 19 日 03 时 14 分 07 秒，便会到达最大时间，过了这个时间点，所有 32 位操作系统时间便会变为 10000000 00000000 00000000 00000000。

32 位 UNIX 系统以 32 位二进制数字表示时间。但是它们最多只能表示至协调世界时间 2038[2]年 1 月 19 日 3 时 14 分 07 秒（二进制 01111111 11111111 11111111 11111111，即 0x7FFF:FFFF），在下一秒二进制数字则会变为 10000000 00000000 00000000 00000000，（即 0x8000:0000）。

0x8000:0000 是一个负数，因此各系统会把时间误解作 1901 年 12 月 13 日 20 时 45 分 52 秒（亦有说回归到 1970 年，即 32 位系统的 UNIX 时间将会被重置），从而出现时间回归的现象，可能导致软件问题和系统瘫痪。

[2]UNIX 系统使用 32 比特的空间存储一个从 1970 年开始数的秒数，这一秒数是 UNIX 内部时钟的基准，但是到 2038 年某一时刻这个空间就不够用了（类似 2000 年的千禧虫电脑问题，被称为 2038 年问题）。

目前的解决方案是把系统由 32 位转为 64 位系统，在 64 位系统下的 UTC 时间最多可以表示到 292,277,026,596 年 12 月 4 日 15 时 30 分 08 秒。

表 31.2: 在不同编程语言中实现 Unix 时间戳到普通时间的转换

语言	操作
Java	`String date = new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm:ss").format(new java.util.Date(Unix timestamp * 1000))`
JavaScript	`var unixTimestamp = new Date(Unix timestamp * 1000)`
	`commonTime = unixTimestamp.toLocaleString()`
Linux	`date -d @Unix timestamp`
MySQL	`from_unixtime(Unix timestamp)`
Perl	`my $time = Unix timestamp`
	`my ($sec, $min, $hour, $day, $month, $year) = (localtime($time))[0,1,2,3,4,5,6]`
PHP	`date('r', Unix timestamp)`
PostgreSQL	`SELECT TIMESTAMP WITH TIME ZONE 'epoch' + Unix timestamp) * INTERVAL '1 second';`
Python	`import time`
	`time.gmtime(Unix timestamp)`
Ruby	`Time.at(Unix timestamp)`
SQL Server	`DATEADD(s, Unix timestamp, '1970-01-01 00:00:00')`
VBScript / ASP	`DateAdd("s", Unix timestamp, "01/01/1970 00:00:00")`
Perl 命令行状态	`perl -e "print scalar(localtime(Unix timestamp))"`

表 31.3: 在不同编程语言中实现普通时间到 Unix 时间戳的转换

语言	操作
Java	`long epoch = new java.text.SimpleDateFormat("dd/MM/yyyy HH:mm:ss").parse("01/01/1970 01:00:00");`

语言	操作
JavaScript	`var commonTime = new Date(Date.UTC(year, month - 1, day, hour, minute, second))`
MySQL	`SELECT unix_timestamp(time)`，时间格式: YYYY-MM-DD HH:MM:SS 或 YYMMDD 或 YYYYMMDD
Perl	`use Time::Local`
	`my $time = timelocal($sec, $min, $hour, $day, $month, $year);`
PHP	`mktime(hour, minute, second, month, day, year)`
PostgreSQL	`SELECT extract(epoch FROM date('YYYY-MM-DD HH:MM:SS'));`
Python	`import time`
	`int(time.mktime(time.strptime('YYYY-MM-DD HH:MM:SS', '%Y-%m-%d %H:%M:%S')))`
Ruby	`Time.local(year, month, day, hour, minute, second)`
SQL Server	`SELECT DATEDIFF(s, '1970-01-01 00:00:00', time)`
Unix / Linux	`date +%s -d"Jan 1, 1970 00:00:01"`
VBScript / ASP	`DateDiff("s", "01/01/1970 00:00:00", time)`

31.2 Leapsecond

The Unix time number is zero at the Unix epoch, and increases by exactly 86400 per day since the epoch.

UNIX 处理闰秒的策略如下:

- 正闰秒时，闰秒的 unix time 就跟后一秒相同;
- 负闰秒时，unix time 就会跳过闰秒的那个数字。

按照 Single Unix specification 的定义，应该是这样的:

```
UTC=2008-12-31T23:59:59.0 → seconds_since_epoch=1230940799[.0]
UTC=2008-12-31T23:59:59.5 → seconds_since_epoch=1230940799[.5]
UTC=2008-12-31T23:59:60.0 → seconds_since_epoch=1230940800[.0]
UTC=2008-12-31T23:59:60.5 → seconds_since_epoch=1230940800[.5]
UTC=2009-01-01T00:00:00.0 → seconds_since_epoch=1230940800[.0]
```

```
UTC=2009-01-01T00:00:00.5 → seconds_since_epoch=1230940800[.5]
```

实际之中有些地方要保证数值的单调增长，会做一些小修改，比如 NTP 协议中采用：

```
UTC=2008-12-31T23:59:59.0 → gettimeofday() returns 1230940799.0
UTC=2008-12-31T23:59:59.5 → gettimeofday() returns 1230940799.5
UTC=2008-12-31T23:59:60.0 → gettimeofday() returns 1230940800.0
UTC=2008-12-31T23:59:60.5 → gettimeofday() returns 1230940800.000001
UTC=2009-01-01T00:00:00.0 → gettimeofday() returns 1230940800.000002
UTC=2009-01-01T00:00:00.5 → gettimeofday() returns 1230940800.5
```

31.3 date()

PHP 的 date() 函数用于格式化一个本地时间/日期的时间戳[3]，从而产生可读性更好的日期和时间。

```
string date( string $format, [, int timestamp] )
```

返回将整数 timestamp 按照给定的格式字串而产生的字符串。如果没有给出时间戳则使用本地当前时间。换句话说，timestamp 是可选的，默认值为 time()。

其中：

- format 是必需的，用于规定时间戳的格式。
- timestamp 是可选的，用于规定时间戳。默认是当前的日期和时间。

date() 函数的第一个参数规定了如何格式化日期/时间，它使用字母来表示日期和时间的格式。下面列出了一些可用的字母：

- d - 月中的天 (01-31)
- m - 当前月，以数字计 (01-12)
- Y - 当前的年（四位数）

通过在字母之间插入其他字符，比如”/”、”.” 或者”-”，这样就可以对日期增加附加格式：

```
<?php
echo date("Y/m/d");
echo "<br />";
echo date("Y.m.d");
echo "<br />";
echo date("Y-m-d");
?>
```

[3]时间戳是自 1970 年 1 月 1 日（00:00:00 GMT）以来的秒数，它也被称为 Unix 时间戳（Unix Timestamp）。

date() 函数的第二个参数规定了一个时间戳，此参数是可选的。如果没有提供时间戳，将使用当前的时间。

31.3.1 timezone

在每次调用日期/时间函数时，如果时区无效则会引发 E_NOTICE 错误，如果使用系统设定值或 TZ 环境变量，则会引发 E_STRICT 或 E_WARNING 消息。

31.3.2 format

自 PHP 5.1.1 起引入了下面的常量作为标准的日期/时间格式来指定 format 参数。

```
DateTime::ATOM
DATE_ATOM
Atom (example: 2005-08-15T15:52:01+00:00)
DateTime::COOKIE
DATE_COOKIE
HTTP Cookies (example: Monday, 15-Aug-2005 15:52:01 UTC)
DateTime::ISO8601
DATE_ISO8601
ISO-8601 (example: 2005-08-15T15:52:01+0000)
DateTime::RFC822
DATE_RFC822
RFC 822 (example: Mon, 15 Aug 05 15:52:01 +0000)
DateTime::RFC850
DATE_RFC850
RFC 850 (example: Monday, 15-Aug-05 15:52:01 UTC)
DateTime::RFC1036
DATE_RFC1036
RFC 1036 (example: Mon, 15 Aug 05 15:52:01 +0000)
DateTime::RFC1123
DATE_RFC1123
RFC 1123 (example: Mon, 15 Aug 2005 15:52:01 +0000)
DateTime::RFC2822
DATE_RFC2822
RFC 2822 (example: Mon, 15 Aug 2005 15:52:01 +0000)
DateTime::RFC3339
```

```
DATE_RFC3339
Same as DATE_ATOM (since PHP 5.1.3)
DateTime::RSS
DATE_RSS
RSS (example: Mon, 15 Aug 2005 15:52:01 +0000)
DateTime::W3C
DATE_W3C
World Wide Web Consortium (example: 2005-08-15T15:52:01+00:00)
```

从 PHP5.1.2 起引入了下面的常量来作为日出时间的返回值格式:

```
SUNFUNCS_RET_TIMESTAMP (integer)
Timestamp
SUNFUNCS_RET_STRING (integer)
Hours:minutes (example: 08:02)
SUNFUNCS_RET_DOUBLE (integer)
Hours as floating point number (example 8.75)
```

31.3.3 request

从 PHP5.1 开始，在 `$_SERVER['REQUEST_TIME']` 中保存发起该请求时刻的 UNIX 时间戳，从而可以统计勤请求的开始时间。

31.3.4 range

有效的时间戳典型范围是格林威治时间 1901 年 12 月 13 日 20:45:54 到 2038 年 1 月 19 日 03:14:07，该范围符合 32 位有符号整数的最小值和最大值。

不过，在 PHP 5.1.0 之前，该范围在某些系统（如 Windows）中限制为从 1970 年 1 月 1 日到 2038 年 1 月 19 日。

31.4 strtotime

要将字符串表达的时间转换成时间戳，应该使用 strtotime()。

- 成功则返回时间戳。
- 不成功则返回 FALSE（在 PHP 5.1.0 之前本函数在失败时返回 -1）。

此外，一些数据库有一些函数将其时间格式转换成时间戳（例如 MySQL 的 UNIX_TIMESTAMP 函数）。

strtotime() 可以将任何英文文本的日期时间描述解析为 UNIX 时间戳，该函数预期接受一个包含美国英语日期格式的字符串并尝试将其解析为 Unix 时间戳（自 January 1 1970 00:00:00 GMT 起的秒数），其值相对于 now 参数给出的时间，如果没有提供此参数则用系统当前时间。

```
int strtotime ( string $time [, int $now = time() ] )
```

- time 日期/时间字符串。
- now 用来计算返回值的时间戳。

strtotime() 使用 TZ 环境变量（如果有的话）来计算时间戳，并且自 PHP 5.1.0 起有更容易的方法来定义时区用于所有的日期/时间函数。

- 在每次调用日期/时间函数时，如果时区无效则会引发 E_NOTICE 错误。
- 如果使用系统设定值或 TZ 环境变量，则会引发 E_STRICT 或 E_WARNING 消息。

有效的时间戳通常从 Fri, 13 Dec 1901 20:45:54 GMT 到 Tue, 19 Jan 2038 03:14:07 GMT（对应于 32 位有符号整数的最小值和最大值），而且不是所有的平台都支持负的时间戳，因此日期范围被限制为不能早于 UNIX 纪元。这意味着在 1970 年 1 月 1 日之前的日期将不能用在 Windows 和某些 Linux 版本等，不过 PHP 5.1.0 及更新的版本克服了此限制。

- 如果给定的年份是两位数字的格式，则其值 0-69 表示 2000-2069，70-100 表示 1970-2000。
- 如果给定的日期格式是 m/d/y，则认为是美国格式。
- 如果给定的日期格式是 d-m-y 或 d.m.y，则认为是欧洲格式。

为了避免日期格式中可能的混淆，最好使用 ISO 8601(YYYY-MM-DD) 格式的日期或者 DataTime::createFromFormat()。

使用 strtotime() 来进行数学计算是不明智的，最好使用 DataTime::add()、DataTime::sub() 或 DateTime::modify() 函数来进行日期的加减等操作。

31.5 strftime

要格式化其它语种的日期，应该用 setlocale() 和 strftime() 函数来代替 date()。

31.6 mktime()

mktime() 函数可为指定的日期返回 Unix 时间戳，因此可以把 date() 和 mktime() 函数结合使用来得到未来或过去的日期。

```
mktime(hour,minute,second,month,day,year,is_dst)
```

夏令时自身的问题也让这种方法比简单地在时间戳上加减一天或者一个月的秒数更可靠。

下面的示例将使用 mktime() 函数为明天创建一个时间戳。

```
<?php
$tomorrow = mktime(0,0,0,date("m"),date("d")+1,date("Y"));
echo "明天是".date("Y/m/d", $tomorrow);
?>
```

PHP date/time 函数允许用户提取并格式化服务器上的日期和时间，当然这些函数依赖于服务器的本地设置（日期/时间函数的行为受到 php.ini 中设置的影响）。date/time 函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

表 31.4: PHP Date/Time 配置选项

名称	默认	描述	可改变
date.default_latitude	“31.7667”	规定默认纬度（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.default_longitude	“35.2333”	规定默认经度（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.sunrise_zenith	“90.83”	规定日出天顶（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.sunset_zenith	“90.83”	规定日落天顶（从 PHP 5 开始可用）。date_sunrise() 和 date_sunset() 使用该选项。	PHP_INI_ALL
date.timezone	“”	规定默认时区（从 PHP 5.1 开始可用）。	PHP_INI_ALL

31.7 PHP Date/Time Functions

表 31.5: PHP Date / Time 函数

函数	描述	PHP
checkdate()	验证格利高里日期。	3
date_default_timezone_get()	返回默认时区。	5
date_default_timezone_set()	设置默认时区。	5
date_sunrise()	返回给定的日期与地点的日出时间。	5

函数	描述	PHP
date_sunset()	返回给定的日期与地点的日落时间。	5
date()	格式化本地时间／日期。	3
getdate()	返回日期／时间信息。	3
gettimeofday()	返回当前时间信息。	3
gmdate()	格式化 GMT/UTC 日期/时间。	3
gmmktime()	取得 GMT 日期的 UNIX 时间戳。	3
gmstrftime()	根据本地区域设置格式化 GMT/UTC 时间/日期。	3
idate()	将本地时间/日期格式化为整数	5
localtime()	返回本地时间。	4
microtime()	返回当前时间的微秒数。	3
mktime()	返回一个日期的 Unix 时间戳。	3
strftime()	根据区域设置格式化本地时间/日期。	3
strptime()	解析由 strftime 生成的日期/时间。	5
strtotime()	将任何英文文本的日期或时间描述解析为 Unix 时间戳。	3
time()	返回当前时间的 Unix 时间戳。	3

31.8 PHP Date/Time Constants

表 31.6: PHP Date / Time 常量

常量	描述	PHP
DATE_ATOM	原子钟格式 (如: 2005-08-15T16:13:03+0000)	
DATE_COOKIE	HTTP Cookies 格式 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_ISO8601	ISO-8601 (如: 2005-08-14T16:13:03+0000)	
DATE_RFC822	RFC 822 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_RFC850	RFC 850 (如: Sunday, 14-Aug-05 16:13:03 UTC)	
DATE_RFC1036	RFC 1036 (如: Sunday, 14-Aug-05 16:13:03 UTC)	
DATE_RFC1123	RFC 1123 (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_RFC2822	RFC 2822 (如: Sun, 14 Aug 2005 16:13:03 +0000)	
DATE_RSS	RSS (如: Sun, 14 Aug 2005 16:13:03 UTC)	
DATE_W3C	World Wide Web Consortium (如:2005-08-14T16:13:03+0000)	

Chapter 32

PHP Filesystem

在物理磁盘的上层的抽象层是文件系统，因此用户的角度来看，数据、信息和程序等都是以文件的形式存储在物理磁盘上。

文件的大小并不是固定的，因此一个文件通常需要对应磁盘上的一个或多个存储单元，目录（或文件夹）可以用来对文件进行有效地区分和管理。

从广义上来说，物理磁盘、文件和目录是操作系统固有地组成部分，通称为文件系统。

32.1 filetype()

PHP 基于 UNIX 系统模型来设计对文件系统的操作，因此 PHP 的文件系统操作函数类似于 UNIX 的 shell 命令。

PHP 将文件划分为 file、dir、block、link、fifo、char 和 unknown 等类型。

Windows 没有提供 UNIX 的文件系统特性，而是通过其自己的实现来对绝大多数函数进行兼容，不过只能支持 file、dir 和 unknown 等文件类型。

file	普通文件
dir	目录
block	设备文件
link	符号链接
fifo	命名管道
char	套接字
unknown	未知类型，使用非 UNIX 结构的文件

32.1.1 is_file()

32.1.2 is_dir()

32.1.3 is_link()

32.1.4 stat()

32.1.5 lstat(

32.1.6 clearstatcache()

PHP 通过缓存 stat()、lstat() 等相关函数的返回信息来提供更快的性能，不过在某些情况下，用户可能需要清除被缓存的信息。

例如，在一个脚本中多次检查同一个文件，而且该文件在脚本执行期间可能被删除或修改，因此需要使用 clearstatcache() 来清除文件状态缓存。

32.2 Operation

文件处理包括文件读、写、删除、锁定，以及检查文件类型、文件完整性等。

为了读取或写入一个本地文件，首先需要以特定方式（只读、只写或可读可写等）打开文件，然后在系统获得对文件的控制权后，就可以执行相应的读写操作。

在文件处理操作结束后，需要及时关闭文件来释放资源。

除了操作本地文件之外，只要激活 php.ini 中的“allow_url_fopen”选项，PHP 也支持访问远程文件。

32.2.1 fopen()

fopen() 函数用于在 PHP 中打开文件。此函数的第一个参数含有要打开的文件的名称，第二个参数规定了使用哪种模式来打开文件：

```
<?php
  $file=fopen("welcome.txt","r");
?>
```

文件可能通过下列模式[1]来打开：

[1]如果 fopen() 无法打开指定文件，则返回 0 (false)。

表 32.1: PHP fopen() 模式

模式	描述
r	只读。在文件的开头开始。
r+	读/写。在文件的开头开始。
w	只写。打开并清空文件的内容；如果文件不存在，则创建新文件。
w+	读/写。打开并清空文件的内容；如果文件不存在，则创建新文件。
a	追加。打开并向文件文件的末端进行写操作，如果文件不存在，则创建新文件。
a+	读/追加。通过向文件末端写内容，来保持文件内容。
x	只写。创建新文件。如果文件已存在，则返回 FALSE。
x+	读/写。创建新文件。如果文件已存在，则返回 FALSE 和一个错误。

如果 fopen() 不能打开指定的文件，下面的例子会生成一段消息：

```
<?php
  $file=fopen("welcome.txt","r") or exit("Unable to open file!");
?>
```

为了使用 fopen() 访问远程文件，需要指定正确的 URL 参数，PHP 可以依据相应的协议和地址进行处理。

PHP 的 fopen() 函数支持 HTTP、HTTPS、FTP 和 FTPS 等协议。

- 对于 HTTP 和 HTTPS，fopen() 只能以只读方式打开 URL；
- 对于 FTP 和 FTPS，fopen() 支持以“只写”或“只读”方式打开 URL；
- 对于 FTP 和 FTPS，fopen() 不能使用“可读可写”方式打开 URL。

在下面的示例中，fopen() 获取远程文件内容后进行分析，并且配合正则表达式进行匹配，从而将结果重新格式化后在页面输出。

```
<?php
  set_time_limit(0);  //为了避免连接超时，这里设定
          //对程序运行时间未做限制
  $remote_url = "http://www.taodoor.com";

  //打开远程文件
  $handdle = fopen($remote_url . "/news.html", "r");

  if($handdle)
  {
    $data = '';
```

```
    //读取文件
    while(!feof($handdle))
    {
      $data .= fgets($handdle, 1024);
    }

    //使用正则表达式分析页面的链接地址
    preg_match_all('/<a\s+href="?([^>"]+)"?\s*[^>]*>([^>]+)<\/a>/i',$data,$arr);

    //循环输出
    for($i=0; $i<count($arr[1]); $i++)
    {
      echo '<li><a href="'.$remote_url.$arr[1][$i].'">'.$arr[2][$i].'</a>';
    }
  }else{
    echo "无法连接到远程服务器。";
  }
?>
```

在访问远程主机并执行文件操作时，难免发生超时错误，因此可以使用 set_time_limit() 函数来对程序的运行时间进行限制。

32.2.2 fclose()

fclose() 函数用于关闭打开的文件。

```
<?php
$file = fopen("test.txt","r");

//some code to be executed

fclose($file);
?>
```

32.2.3 fsize()

32.2.4 feof()

feof() 函数检测是否已达到文件的末端 (EOF)。在循环遍历未知长度的数据时，feof() 函数很有用，但是在 w 、 a 以及 x 模式下无法读取打开的文件。

```
if (feof($file)) echo "End of file";
```

32.2.5 fread()

下面的示例代码说明了如何正确地将文件内容读取到 $content 字符串中，并且对于二进制文件同样也是安全的。

```
<?php
  $filename = "/usr/local/readme.txt";

  //一次性读取整个文件
  $handle = fopen ($filename, "r");   //打开一个只读文件
  $length = filesize ($filename);    //计算文件的大小
  $content = fread ($handle, $length);   //读取文件的内容
  fclose ($handle);       //关闭文件

  //传统读取文件的方法
  $handle = fopen ($filename, "r+");
  $content = "";

  //使用feof()判断文件的结束
  while(!feof($handle))
  {
    $content .= fread($handle, 1024);
  }
  fclose ($handle);

  //更便捷的方法
  $handle = fopen ($filename, "r");
  $content = "";
  do{
    $data = fread($handle, 1024);
    if(strlen($data)===0)     //当没有数据时，跳出循环
      break;
    $content .= $data;
  }while(1);
  fclose ($handle);

  //使用fgets()的方法
  $handle = fopen ($filename, "rt");   //使用“t”将“\n”替换为“\r\n”
  while (!feof ($handle))
  {
    $content .= fgets($handle, 4096);
  }
```

```
  fclose ($handle);

  //使用fgetc()逐字节读取文件内容
  $handle = fopen ($filename, "r");
  while (($c = fgetc($handle))!==FALSE)
  {
    $content .= $c;
  }
  fclose ($handle);
?>
```

32.2.6 fgets()

fgets() 函数用于从文件中逐行读取文件。在调用该函数之后，文件指针会移动到下一行。

下面的例子逐行读取文件，直到文件末端为止：

```
$file = fopen("welcome.txt", "r") or exit("Unable to open file!");
//Output a line of the file until the end is reached
while(!feof($file))
  {
  echo fgets($file). "<br />";
  }
fclose($file);
?>
```

32.2.7 fgetc()

fgetc() 函数用于从文件逐字符地读取文件。在调用该函数之后，文件指针会移动到下一个字符。

下面的例子逐字符地读取文件，直到文件末端为止：

```
$file = fopen("welcome.txt", "r") or exit("Unable to open file!");
//Output a line of the file until the end is reached
while(!feof($file))
  {
  echo fgetc($file). "<br />";
  }
fclose($file);
?>
```

32.2.8 fwrite()

下面的示例代码说明如何写入文件。

```
<?php
  $filename = 'test.txt';
  $somecontent = "添加这些文字到文件\n";

  //首先要确定文件存在并且可写
  if (is_writable($filename))
  {
       //使用添加模式打开$filename, 文件指针将会在文件的开头
      if (!$handle = fopen($filename, 'a'))
    {
      echo "不能打开文件 $filename";
      exit;               //无法打开文件则退出程序
        }

       //将$somecontent写入到我们打开的文件中。
    if (!fwrite($handle, $somecontent))
    {
      echo "不能写入到文件 $filename";
      exit;               //无法写入文件则退出程序
    }

    echo "文件“$filename”写入成功";

    //关闭文件
    fclose($handle);
  } else {
    echo "文件“$filename”不可写";
  }
?>
```

在写入文件之前需要使用 is_writable() 函数判断文件是否可写，然后在可写的情况下按照打开、写入、关闭文件的顺序依次进行操作。

不同的操作系统使用不同的行结束符，因此在写入一个文本文件并需要插入一个新行时需要使用符合相应操作系统的行结束符。

- UNIX/Linux 等系统使用“\n”作为行结束符；
- Windows 系统使用“\r\n”作为行结束符。
- Mac OS X 操作系统使用“\r”作为行结束符。

32.2.9 fputs()

32.2.10 unlink()

在删除文件时可能会发生删除失败的情况，其原因可能包括文件不存在、文件权限错误、文件被锁定以及删除目录而非文件等。

为了屏蔽删除文件失败时发生的错误，通常可以使用“@”来禁止显示删除文件时发生的 E_WARNING 错误报告。

32.2.11 ftruncate()

32.2.12 file_get_contents()

fopen()、fread()、fgets()、fseek() 等只能对文件进行最基本的处理，并返回文件指针。

file_get_contents() 和 file() 函数可以使用更加简洁的方式来获得文件的全部或部分内容，并返回字符串或数组。

file_get_contents() 函数可以将整个文件读入到一个字符串中，并且避免文件的打开、锁定和关闭等操作。

在操作系统提供支持的情况下，file_get_contents() 函数还可以使用内存映射技术来增强性能。

和 fopen() 等函数打开本地或远程文件连接类似，file_get_contents() 函数除了可以读取本地文件之外，还可以读取远程主机上的文件。

```
<?php
 //读取一个本地文件
 echo file_get_contents("/home/tom/public_html/index.html");

 //读取一个远程文件
 echo file_get_contents("http://www.taodoor.com/index.php");

 //如果使用低版本的PHP程序，应该使用代替方案，
 //这样可以增强程序的移植性
 if(!file_exists("file_get_contents"))
 {
   //自定义的代替方法
   function file_get_contents ($filename)
   {
     //打开文件
     $fp = @fopen ($filename, "r");

     //锁定文件
```

```
    @flock($fp, LOCK_SH);

    //读取文件内容
    $data = @fread($fp, filesize($filename));

    //关闭文件
    @fclock($fp);

    if($data)
    {
      return $data;
    }else{
      return "";
    }
   }
  }
?>
```

32.2.13 file()

file() 函数可以将整个文件读入到一个数组中，这样就可以使用数组的相关函数来对文件内容进行处理。

从实质上来讲，file() 函数通过把整个文件进行分割后读入数组，这样数组中的每个元素就是文件中相应的行（并且包括换行符在内）。

在分析日志文件时，可以通过 file() 函数实现对日志文件内容倒序处理，这样就可以从日志文件中获得最新的记录，它们原来的位置是日志文件的尾部。

```
<?php
  $logs = file("server.log");   //读取一个日志文件
  $contents = array_reverse($logs);
  foreach($contents as $line){
    echo $line;
  }
?>
```

32.3 Pointer

在 PHP 中对文件指针进行操作时，必须首先提供一个使用 fopen() 函数打开的、合法的文件指针。

文件指针的位置是以文件头开始的字节数度量的。

32.3.1 ftell()

32.3.2 fseek()

32.3.3 rewind()

32.4 Tempfile

PHP 支持创建一个临时文件来存储数据，使用 tmpfile() 和 tempnam() 可以建立一个具有唯一文件名的临时文件。

通常情况下，PHP 文件函数创建的临时文件位于系统临时目录/tmp 下。

32.4.1 tmpfile()

tmpfile() 函数不需要任何参数就可以以写模式建立一个具有唯一文件名的临时文件，并返回一个和 fopen() 相似的文件指针。

tmpfile() 创建的临时文件在 fclose() 关闭文件或脚本执行结束后会被自动删除。

32.4.2 tempnam()

- tmpfile() 无法指定临时文件的目录，也无法获知创建的文件名。
- tempnam() 可以对临时文件提供更多的控制。

tempnam() 函数可以在指定的目录下创建一个具有唯一文件名的文件，而且 tempnam() 创建的临时文件不会被自动删除，除非使用 unlink() 进行手动删除。

32.5 Lock

32.5.1 flock()

在网络环境下，不同用户进程可能在同一时刻对同一文件进行操作，不合理的并发模型可能会对文件造成破坏。

最典型的网络文件访问破坏是多个进程同时对文件进行写入操作，用户可以使用 flock() 来对文件使用锁机制来避免文件访问破坏。

flock() 允许执行一个简单的可以在任何平台中使用的读取/写入模型（包括大部分的 Unix 派生版和 Windows）。

```
bool flock ( resource $handle , int $operation [, int &$wouldblock ] )
```

flock() 函数获取文件的指针 $handle，并将该文件设定为某一访问权限 $operation。

$operation 可以是以下值之一：

- LOCK_SH：共享锁定，读取文件时使用；
- LOCK_EX：独占锁定，写入文件时使用；
- LOCK_UN：释放锁定，释放已经存在的共享或独占锁；
- LOCK_NB：附加锁定，如果不希望 flock() 在锁定时堵塞，就应该在上述锁定后加上该锁。

PHP 支持以咨询方式对全部文件执行轻便的锁定，也就是说所有访问程序必须使用同一方式锁定, 否则它不会工作。

flock() 函数可以辨别同时运行的独立进程，判断它们能否在共享或独占模式下访问文件。

默认情况下，flock() 函数会阻塞到获取锁，可以通过 LOCK_NB 选项来控制（在非 Windows 平台上）。

如果其他进程不服从或不识别 flock() 函数加在用户文件上的锁，那么这个进程会忽略已存在的文件锁，因此只有所有访问程序使用相同的方式锁定文件时才能让 flock() 发挥作用，这样就要求其他访问程序都必须服从于 flock() 函数。

一个文件可以同时存在很多共享锁定 LOCK_SH，这样多个进程可以在同一时刻拥有对该文件的读取访问权限；

一个独占锁定 LOCK_EX 只允许一个用户拥有一次，通常被用于文件的写入操作。

如果其他进程需要访问具有独占锁定的文件，那么必须等到独占锁定被释放以后才能进行。

```
<?php
  $fname = "/home/tom/public_html/test.txt";

  //以读写方式打开文件
  if( $fp = @fopen($fname, "w+") )
  {
    die("无法打开文件 $fname");
  }

  //写入文件时，总将文件锁定为LOCK_EX
  if( !flock($fp, LOCK_EX) )
  {
    die("无法将文件锁定为 LOCK_EX\n");
  }
  //写入文件
  fwrite($fp, "第一行文字\n");
  fwrite($fp, "另一行文字\n");

  flock($fp, LOCK_UN);        //释放独占锁定
```

```
  //读取文件时，总将文件锁定为LOCK_SH
  if( !flock($fp, LOCK_SH) )
  {
    die("无法将文件锁定为 LOCK_SH\n");
  }

  //读取文件
  rewind($fp);              //将文件指针倒回文件的开头
  while(!feof($fp))
  {
    echo fread($fp, 1024);
  }

  flock($fp, LOCK_UN);        //释放共享锁定

  fclose($fp);              //关闭文件
?>
```

在 Windows 以外的操作系统中，对于已经被 flock() 函数锁定的文件，如果再次执行锁定就会使 flock() 函数被挂起（锁定堵塞），直到以前的所有其他锁定被释放。

如果不希望上述锁定堵塞的情况发生，可以在 $operation 参数中附加 LOCK_NB。

下面的示例说明了如何防止锁定堵塞发生。

```
<?php
  $fp = fopen($fname, "w+");
  flock($fp, LOCK_EX + LOCK_NB);
  fwrite($fp, "Some thing wrote\n");
  flock($fp, LOCK_UN + LOCK_NB);
  fclose();
?>
```

32.6 Upload

PHP 允许用户从表单上传文件到服务器，也不限制浏览器上传的文件类型，而且 PHP 还允许用户对服务 iq 文件的下载进行控制。

不过，从安全方面来考虑时会发现允许用户上传文件是一个巨大的安全风险，因此应该仅仅允许可信任的用户执行文件上传操作。

最基本的文件上传方式是使用 HTML 表单的 POST 方法来提交文件。

```
<!DOCTYPE html>
<html>
<body>

<form action="upload_file.php" method="post" enctype="multipart/form-data">
<input type="hidden" name="MAX_FILE_SIZE" value="40000">
<label for="file">Filename:</label>
<input type="file" name="file" id="file" />
<br />
<input type="submit" name="submit" value="Submit" />
</form>

</body>
</html>
```

在隐藏域的 MAX_FILE_SIZE 中可以设置允许接收文件的最大尺寸（单位为字节），但是 MAX_FILE_SIZE 的值只是对浏览器的一个建议，可以被简单的绕过。

在实际应用中，php.ini 中设置的文件上传的最大值 upload_max_filesize 是不会失效的，不过最好还是在表单中通过 MAX_FILE_SIZE 来进行限制。

<form> 标签的 enctype 属性规定了在提交表单时要使用哪种内容类型。在表单需要二进制数据时，比如文件内容，使用 “multipart/form-data”。

<input> 标签的 type=”file” 属性规定了应该把输入作为文件来处理。举例来说，当在浏览器中预览时，会看到输入框旁边有一个浏览按钮。

通过使用 PHP 的全局数组 $_FILES，用户可以从客户计算机向远程服务器上传文件，下面就是 “upload_file.php” 文件中用于上传文件的代码：

```
<?php
if ($_FILES["file"]["error"] > 0)
  {
  echo "Error: " . $_FILES["file"]["error"] . "<br />";
  }
else
  {
  echo "Upload: " . $_FILES["file"]["name"] . "<br />";
  echo "Type: " . $_FILES["file"]["type"] . "<br />";
  echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
  echo "Stored in: " . $_FILES["file"]["tmp_name"];
  }
?>
```

其中，第一个参数是表单的 input name，第二个下标可以是 “name”,“type”, “size”, “tmp_name”

或 “error”。就像这样：

- `$_FILES["file"]["name"]` - 被上传文件的名称
- `$_FILES["file"]["type"]` - 被上传文件的类型
- `$_FILES["file"]["size"]` - 被上传文件的大小，以字节计
- `$_FILES["file"]["tmp_name"]` - 存储在服务器的文件的临时副本的名称
- `$_FILES["file"]["error"]` - 由文件上传导致的错误代码

在上述的简单文件上传实现中，服务器接收到的用户文件首先被存储在临时目录中，PHP 将获得一个存储了上传文件的全部信息的 $_FILES 全局变量。

在处理文件上传的 PHP 脚本中，需要实现对文件进行检测并判断的逻辑，然后才能确定接下来是否允许文件开始上传。

- `$_FILES["file"]["size"]` 限制被上传文件的大小；
- `$_FILES["file"]["type"]` 限制被上传文件的类型；
- `$_FILES["file"]["error"]` 判断文件上传导致的错误代码。

如果上传后的文件大小、类型都符合限制，那么接下来 PHP 可以将其从临时目录拷贝到指定的位置，从而完成文件上传。

基于安全方面的考虑，开发者应当增加有关什么用户有权上传文件的限制，因此需要增加对文件上传的限制，例如只允许用户上传.gif 或.jpeg 文件[2]，文件大小必须小于 20 kb。

```
<?php

if ((($_FILES["file"]["type"] == "image/gif")
|| ($_FILES["file"]["type"] == "image/jpeg")
|| ($_FILES["file"]["type"] == "image/pjpeg"))
&& ($_FILES["file"]["size"] < 20000))
  {
  if ($_FILES["file"]["error"] > 0)
    {
    echo "Error: " . $_FILES["file"]["error"] . "<br />";
    }
  else
    {
    echo "Upload: " . $_FILES["file"]["name"] . "<br />";
    echo "Type: " . $_FILES["file"]["type"] . "<br />";
    echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
    echo "Stored in: " . $_FILES["file"]["tmp_name"];
    }
  }
else
```

[2]对于 IE，识别 jpg 文件的类型必须是 pjpeg，对于 FireFox，必须是 jpeg。

```
  {
  echo "Invalid file";
  }

?>
```

上面的例子在服务器的 PHP 临时文件夹创建了一个被上传文件的临时副本，但是这个临时的复制文件会在脚本结束时消失。

32.6.1 move_uploaded_file()

要保存被上传的文件，我们需要把它拷贝到另外的位置，PHP 提供了专门用于上传文件拷贝的函数 move_uploaded_file()，这样就有了 copy() 函数之外的另一种选择。

```
<?php
if ((($_FILES["file"]["type"] == "image/gif")
|| ($_FILES["file"]["type"] == "image/jpeg")
|| ($_FILES["file"]["type"] == "image/pjpeg"))
&& ($_FILES["file"]["size"] < 20000))
  {
  if ($_FILES["file"]["error"] > 0)
    {
    echo "Return Code: " . $_FILES["file"]["error"] . "<br />";
    }
  else
    {
    echo "Upload: " . $_FILES["file"]["name"] . "<br />";
    echo "Type: " . $_FILES["file"]["type"] . "<br />";
    echo "Size: " . ($_FILES["file"]["size"] / 1024) . " Kb<br />";
    echo "Temp file: " . $_FILES["file"]["tmp_name"] . "<br />";

    if (file_exists("upload/" . $_FILES["file"]["name"]))
      {
      echo $_FILES["file"]["name"] . " already exists. ";
      }
    else
      {
      move_uploaded_file($_FILES["file"]["tmp_name"],
      "upload/" . $_FILES["file"]["name"]);
      echo "Stored in: " . "upload/" . $_FILES["file"]["name"];
      }
    }
```

```
  }
else
  {
  echo "Invalid file";
  }
?>
```

这个例子把用户上传的文件保存到名为“upload”的新文件夹，上面的脚本会检测是否已存在此文件，如果不存在，则把文件拷贝到指定的文件夹。

从思路上来看，处理多个文件的上传和单个文件的上传的情况是一样的，因此表单可以支持更多数量的文件上传，区别只是 $_FILES 数组的结构形式。

首先，在上传单个文件时，需要在 <input> 标签中指定文件的名字，并且在处理文件上传的脚本中对其进行处理。

```
<form enctype="multipart/form-data" method="POST" action="upload.php">
  <input type="hidden" name="MAX_FILE_SIZE" value="40000">
  文件: <input name="upfile" type="file">
  <input type="submit" value="上传文件">
</form>

<?php
  //上传文件的存储目录
  $uploaddir = "/home/tom/public_html/uploads/";

  //文件上传后全路径名称
  $uploadfile = $uploaddir. $_FILES["upfile"]["name"];

  if (move_uploaded_file($_FILES["upfile"]["tmp_name"], $uploadfile))
  {
     print "文件上传成功! \n";
     print_r($_FILES);
  } else {
     print "文件上传失败! \n";
     print_r($_FILES);
  }
?>
```

其次，在上传多个文件时，需要在 <input> 标签中指定不同的 name 属性值。

```
<form enctype="multipart/form-data" method="POST" action="multiy_upload.php">
  文件1: <input name="upfile1" type="file">
  文件2: <input name="upfile2" type="file">
  文件3: <input name="upfile3" type="file">
```

```
  <input type="submit" value="上传文件">
</form>

<!--
<form enctype="multipart/form-data" method="POST" action="multiy_upload.php">
  <input type="hidden" name="MAX_FILE_SIZE" value="40000">
  文件1: <input name="upfile[]" type="file"><br>
  文件2: <input name="upfile[]" type="file"><br>
  文件3: <input name="upfile[]" type="file"><br>
  <input type="submit" value="上传文件">
</form>
-->

<?php
  //文件上传目录
  $uploaddir = "/home/tom/public_html/uploads/";

  //循环遍历$_FILE数组进行判断
  for($i=0; $i<count($_FILES['upfile']['name']); $i++)
  {
    //错误数组
    $error = array();

    //判断文件大小
    if($_FILES['upfile']['size'][$i] >= 40000)
    {
      $error[] = "文件的尺寸太大! ";
    }

    //判断文件类型
    if($_FILES['upfile']['type'][$i] != "text/plain")
    {
      $error[] = "文件的类型必须为文本文件! ";
    }

    //其他错误类型
    if($_FILES['upfile']['error'][$i] != UPLOAD_ERR_OK)
    {
      $error[] = "文件上传失败! ";
    }
```

```
    echo "文件{$i}: ".$_FILES['upfile']['name'][$i]."<br>\n";

    if(count($error))
    {
      //发现错误
      echo join("<br>", $error);
    }else{
      move_uploaded_file($_FILES['upfile']['tmp_name'][$i],
            $uploaddir . $_FILES['upfile']['name'][$i]);
    }
    echo "<hr>";
  }
?>
```

32.7 Download

要实现文件下载，必须首先向浏览器发送必要的头信息来通知浏览器准备对将要下载的文件的处理。

Web 服务器向浏览器发送的信息主要包括以下内容：

- 下载文件的类型使用 MIME 类型表示；
- 下载文件的描述给出文件名等信息；
- 下载文件的长度以字节为单位进行说明。

Web 服务器向浏览器发送的最常见的的 MIME 类型就是“Content-type: text/html”。如果 Web 服务器向浏览器发送的是 PNG 图像，那么 HTTP 头信息可能如下：

```
Content-type: image/png
Content-Transfer-Encoding: BINARY
```

在不同的系统中，MIME 类型的表示根据 Web 服务器的配置并不一致，例如可以在 Apache httpd 服务器的 conf 目录下查看 magic 文件来检查 MIME 类型。

32.7.1 header()

header() 函数可以向浏览器发送头信息来指导下载，而且头信息必须在其他页面内容输出之前被发送，否则发生错误。

在 header() 函数发送完头信息后，可以使用 echo()、print() 等发送具体的 HTML 信息等输出到浏览器。

```
<?php
  //文件源是: original.html
```

```
  $name = 'original.html';

  //发送头信息: 指定文件类型
  header('Content-type: text/html');

  //发送头信息: 指定文件的表述
  header('Content-Disposition: attachment; filename="downloaded.html"');

  //发送头信息: 指定文件的小
  header("Content-Length: ".filesize($name));

  //输出文件内容
  $fp = fopen($name, "r");
  while(false==feof($fp)){
    echo fread($fp, 1024);
  }
?>
```

32.7.2 readfile()

在输出文件内容时可以使用 readfile() 函数来将文件的内容直接输出到浏览器。

```
<?php
  //一个图片文件
  $name = "img.png";

  //发送头信息
  header("Content-Type: image/png");
  header("Content-Disposition: attachment; filename=\"$name\"");
  header("Content-Length: ".filesize($name));

  //输出文件
  readfile($fp);
?>
```

32.8 Permission

UNIX 系统在安全性方面被设计为一个分级的权限控制系统，任何系统资源都属于特定的用户或用户组，其他用户或用户组在访问时则受到相应的限制。

超级用户 root 拥有对所有资源的绝对的控制权。

- 只有 root 用户可以任意修改文件的所有者和用户组；
- 其他用户只能将文件的组改为该用户自己所在的组，不能修改文件所有者。

需要注意的是，访问权限是 UNIX/Linux 系统的一种内部机制，相关的函数在 Windows 操作系统下无效。

32.8.1 fileowner()

通常情况下，系统资源（即文件或目录）属于其创建者，fileowner() 和 filegroup() 可以返回文件所有者 ID 和所属的用户组 ID。

32.8.2 filegroup()

32.8.3 posix_getpwuid()

posix_getpwuid() 和 posix_getgrgid() 可以将用户 ID 和用户组 ID 解析为字符串形式。

32.8.4 posix_getgrgid()

32.8.5 chown()

32.8.6 chgrp()

实际上，PHP 运行在非 root 用户状态下，因此 chgrp() 函数在修改文件所属的组时受到一定的限制，而且使用 chown() 函数改变文件的所有者时不会生效。

32.8.7 chmod()

文件的访问权限可以用来指定特定的用户能否对文件进行读、写或执行等操作。

chmod() 函数可以改变文件的访问权限，需要指定一个文件和相对应的权限值。

默认情况下，权限值需要使用以 0 开头的八进制数字表示。

32.9 Path

为了描述一个文件的位置，需要给出文件的路径和文件名。

PHP 支持两种不同的路径分隔符，其中：

- UNIX/Linux 系统必须使用“/”；
- Windows 默认使用“\”，也接受“/”的写法。

在考虑可移植情况时，建议使用“/”作为文件的路径分隔符，或者使用 PHP 内置的 DIRECTORY_SEPARATOR，其值为当前系统的默认路径分隔符。

32.9.1 basename()

32.9.2 dirname()

32.9.3 pathinfo()

pathinfo() 返回一个关联数组包含有 path 的信息。返回关联数组还是字符串取决于 options。

pathinfo() 返回的数组包含了路径的信息，其中的 dirname、basename 和 extension 等分别表示目录名、文件名和文件扩展名。

32.10 Directory

PHP 提供了与目录操作相关的函数，也可以使用目录类实现对文件系统中的目录的操作。

如果用户需要取得一个目录下的文件和子目录，可以使用遍历目录和检索目录两种方式来实现。

读取目录内容时可以使用 opendir()、readdir() 和 closedir() 等函数，类似于读取文件的操作，因此也称为遍历目录。

- 遍历目录结构时可以使用文件系统函数，也可以使用 Directory 类；
- 检索目录可以返回包含目录结构信息的数组。
- opendir() 打开一个目录句柄，可用于之后的 closedir()，readdir() 和 rewinddir() 调用中。
- readdir() 返回目录中下一个文件的文件名。文件名以在文件系统中的排序返回。
- closedir() 关闭由 dir_handle 指定的目录流。流必须之前被 opendir() 所打开。

下面的示例说明如何使用 opendir()、readdir() 和 closedir() 等实现遍历目录结构。

```
<?php
 //打开一个目录
 $handle = @opendir("/path/to/files");

 if ($handle)
    {
   echo "目录文件: \n";
```

```
    readdir($handle);  //获取“.”，当前目录的表示
    readdir($handle);  //获取“..”，上级目录的表示

    //正确地遍历目录方法，使用“!==”符号
    while (false !== ($file = readdir($handle))) {
      echo "$file\n";
    }

    //将目录的指针倒回开头
    rewinddir($handle);

    //关闭目录
    closedir($handle);
  }
?>
```

PHP 提供的内置的 Directory 类可以使用面向对象的方式来实现目录遍历。

```
Directory {
  /* 属性 */
  public string $path ;
  public resource $handle ;
  /* 方法 */
  public void close ([ resource $dir_handle ] )
  public string read ([ resource $dir_handle ] )
  public void rewind ([ resource $dir_handle ] )
}
```

Directory 类是 PHP 的内建类，没有提供相应的构造函数，因此 Directory 实例是通过调用 dir() 函数创建的，而不是 new 操作符。

下面的示例说明如何使用 Directory 类来实现目录遍历。

```
<?php
  //打开目录
  $d = dir("/path/to/files");

  echo "Handle: ".$d->handle."<br>\n";
  echo "Path: ".$d->path."<br>\n";

  //循环输出
  while (false !== ($file = $d->read()))
  {
     echo $file."<br>\n";
```

```
 }

 //关闭目录
 $d->close();
?>
```

具体来说，dir() 函数可以返回一个实例化了的 Directory 对象，其中：

- $handle 是一个打开了的目录句柄；
- $path 是当前目录的路径。

Directory 类的 read()、rewind() 和 close() 方法分别对应 readdir()、rewinddir() 和 closedir() 函数。

32.10.1 opendir()

opendir() 函数接受一个目录路径字符串，并返回一个目录句柄。如果目录不存在或者无访问权限，则返回 FALSE，同时产生一个 E_WARNING 级别的错误信息。

在 opendir() 前面加上“@”符号可以抑制错误信息的输出。

32.10.2 readdir()

readdir() 函数接受一个目录句柄，并返回当前目录指针位置的一个文件名，然后将目录指针向后移动一位，直到目录指针位于目录的末尾时才会返回 FALSE 以表示再没有文件了。

如果要重置目录指针到开始处，可以使用 rewinddir() 函数，该函数可以接受一个目录句柄，并将其倒回目录的开头。

32.10.3 closedir()

closedir() 函数关闭一个打开的目录，也可以接受一个目录句柄作为参数来关闭指定的目录。

32.10.4 glob()

如果用户使用 glob() 函数检索指定的目录来实现遍历目录，也可以实现和其他目录函数类似的效果。

glob() 函数依照 libc glob() 函数使用的规则寻找所有与 pattern 匹配的文件路径，类似于一般 shells 所用的规则一样，不进行缩写扩展或参数替代。

glob() 函数的返回值是一个包含目录检索结果的数组，其原型如下：

```
array glob ( string $pattern [, int $flags = 0 ] )
```

- $pattern 指定要检索的目录信息，可以使用“*”或“?”等通配符；
- $flags 指定与检索模式相关的参数。

下面的示例说明如何使用 glob() 函数实现检索目录文件。

```
<?php
  //检索当前目录中的所有“*.txt”文件
  $files = glob("*.txt");
  foreach ($files as $filename)
  {
      echo "$filename size " . filesize($filename) . "\n";
  }

  //检索当前目录中的所有以“c”开头的子目录
  $files = glob("c*", GLOB_ONLYDIR);
  foreach ($files as $filename)
  {
      echo "$filename size " . filesize($filename) . "\n";
  }

  //检索“/path/”中的所有以“a”、“b”或“c”开头的PHP文件
  $files = glob("/path/{a,b,c}*.php", GLOB_BRACE);
  foreach ($files as $filename)
  {
      echo "$filename size " . filesize($filename) . "\n";
  }
?>
```

32.10.5 mkdir()

32.10.6 rmdir()

通常情况下，如果目录是非空的，那么就不能进行快速的删除，必须首先删除其中的文件，然后才能从最深处的目录开始层层向外删除目录。

递归思想的引入使得可以一次删除整个目录及其子目录和其中的文件。

首先，考虑要删除的目录中没有包含其他子目录的情况，只要遍历该目录并删除其他的所有文件后，就可以实现目录删除。

其次，如果目录中还包含子目录，那么就使用递归方式来删除所有目录。

```
<?php
  /**
  * 函数名：deleteDir
```

```
 * 功 能：递归地删除指定的目录
 * 参 数：$dir目录
 * 返回值：无
   */
 function deleteDir ($dir)
 {
   $handle = @opendir ($dir); //打开目录

   readdir ($handle);   //排除当前目录“.”
   readdir ($handle);   //排除父级目录“..”
   while (false !== ($file = readdir($handle)))
   {
     //构造文件或目录的路径
     $file = $dir .DIRECTORY_SEPARATOR. $file;

     if (is_dir ($file)){   //如果是子目录，就进行递归操作
       delete ($file);
     } else {      //如果是文件，则使用unlik()删除
       if (@unlink ($file)) {
         echo "文件<b>$file</b>删除成功了。<br>\n";
       } else {
         echo "文件<b>$file</b>删除失败！<br>\n";
       }
     }
   }

   //现在，删除当前目录
   if (@rmdir ($dir)) {
     echo "目录<b>$dir</b>删除成功了。<br>\n";
   } else {
     echo "目录<b>$dir</b>删除失败！<br>\n";
   }
 }

 //测试程序
 $dir = "/home/tom/public_html/delete_dir";
 deleteDir ($dir);
?>
```

在实际应用中，还需要考虑给定的目录是否已被删除，因此需要通过返回的状态值进行判断，或者在无法删除时终止并返回等。

32.10.7 copy()

PHP 没有提供直接复制目录的函数，需要通过文件拷贝、删除、重命名等方法间接实现。

32.10.8 rename()

重命名文件函数 rename() 对目录操作也是有效的。

从表面上看，移动一个文件是将文件从一个目录复制到另一个目录，然后再删除原来目录中的文件。

从实现原理上看，移动一个文件也是一个重命名的过程，也就是将文件名中的路径名修改为移动文件后的新路径名，从而可以使用重命名思想来简化文件移动。

```
<?php
  /**
  * 函数名：move
  * 功  能：移动指定的文件和目录
  * 参  数：$source 要操作的文件
  *    $dest   要移动到的文件目录
  * 返回值：bool
     */
  function move ($source, $dest)
  {
    $file = basename($source);   //取得目录名
    $desct = $desct .DIRECTORY_SEPARATOR. $file;

    return rename($source, $desct);
  }

  //将文件或目录从path1移动到path2目录下
  move("/path1/name.gif", "/path2");
  move("/path1/dir", "/path2");
?>
```

复制一个包含多级子目录的目录与删除一个非空目录时的情况类似，都需要涉及到文件复制、目录创建等操作，因此也是一个目录遍历和递归处理的综合。

首先，对源目录进行遍历。

- 如果遇到的是文件，可以直接使用 copy() 函数进行复制；
- 如果遇到的是目录，则必须先建立该目录，然后对其下的文件或子目录进行复制操作。

其次，使用递归思想处理整个目录中的所有文件和子目录。

```
<?php
```

```
/**
* 函数名：copyDir
* 功　能：递归地复制整个目录
* 参　数：$dirFrom 源目录名
*　 $dirTo　 目标目录名
* 返回值：无
  */
function copyDir($dirFrom, $dirTo)
{
  //如果遇到一个同名的文件，无法复制
  //目录则直接退出
  if(is_file($dirTo))
  {
    die("无法建立目录 $dirTo");
  }

  //如果目录已经存在就不必建立
  if(!file_exists($dirTo))
  {
    mkdir($dirTo);
  }

  $handle = opendir($dirFrom);  //打开当前目录

  readdir ($handle);     //排除当前目录“.”
  readdir ($handle);     //排除父级目录“..”

  //循环读取文件
  while (false !== ($file = readdir($handle)))
  {
    //生成源文件名
    $fileFrom = $dirFrom .DIRECTORY_SEPARATOR. $file;
    //生成目标文件名
    $fileTo = $dirTo .DIRECTORY_SEPARATOR. $file;

    if(is_dir($fileFrom)){   //如果是子目录，就进行递归操作
      copyDir($fileFrom, $fileTo);
    }else{        //如果是文件，直接使用copy()函数
      @copy($fileFrom, $fileTo);
    }
  }
```

```
 }

 //测试代码
 copyDir("C:\\windows\\temp", "D:\\temp");
?>
```

32.11 PHP Filesystem Functions

Filesystem 函数允许您访问和操作文件系统。Filesystem 函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

文件系统函数的行为受到 php.ini 中设置的影响。

表 32.2: PHP Filesystem 配置选项

名称	默认	描述	可改变
allow_url_fopen	“1”	本选项激活了 URL 形式的 fopen 封装协议使得可以访问 URL 对象例如文件。 默认的封装协议提供用 ftp 和 http 协议来访问远程文件，一些扩展库例如 zlib 可能会注册更多的封装协议。 PHP 4.0.4 版以后可用。	PHP_INI_SYSTEM
user_agent	NULL	定义 PHP 发送的 User-Agent。 PHP 4.3.0 版以后可用。	PHP_INI_ALL
default_socket_timeout	“60”	基于 socket 的流的默认超时时间 (秒)。 PHP 4.3.0 版以后可用。	PHP_INI_ALL
from	“”	定义匿名 ftp 的密码 (email 地址)。	PHP_INI_ALL

名称	默认	描述	可改变
auto_detect_line_endings	“0”	当设为 On 时，PHP 将检查通过 fgets() 和 file() 取得的数据中的行结束符号是符合 Unix，MS-DOS，还是 Macintosh 的习惯。 这使得 PHP 可以和 Macintosh 系统交互操作，但是默认值是 Off，因为在检测第一行的 EOL 习惯时会有很小的性能损失，而且在 Unix 系统下使用回车符号作为项目分隔符的人们会遭遇向下不兼容的行为。 PHP 4.3.0 版以后可用。	PHP_INI_ALL

当在 Unix 平台上规定路径时，正斜杠 (/) 用作目录分隔符。而在 Windows 平台上，正斜杠 (/) 和反斜杠 (\) 均可使用。

表 32.3: PHP Filesystem 函数

函数	描述	PHP
basename()	返回路径中的文件名部分。	3
chgrp()	改变文件组。	3
chmod()	改变文件模式。	3
chown()	改变文件所有者。	3
clearstatcache()	清除文件状态缓存。	3
copy()	复制文件。	3
delete()	参见 unlink() 或 unset()。	
dirname()	返回路径中的目录名称部分。	3
disk_free_space()	返回目录的可用空间。	4
disk_total_space()	返回一个目录的磁盘总容量。	4
diskfreespace()	disk_free_space() 的别名。	3
fclose()	关闭打开的文件。	3
feof()	测试文件指针是否到了文件结束的位置。	3
fflush()	向打开的文件输出缓冲内容。	4

函数	描述	PHP
fgetc()	从打开的文件中返回字符。	3
fgetcsv()	从打开的文件中解析一行，校验 CSV 字段。	3
fgets()	从打开的文件中返回一行。	3
fgetss()	从打开的文件中读取一行并过滤掉 HTML 和 PHP 标记。	3
file()	把文件读入一个数组中。	3
file_exists()	检查文件或目录是否存在。	3
file_get_contents()	将文件读入字符串。	4
file_put_contents()	将字符串写入文件。	5
fileatime()	返回文件的上次访问时间。	3
filectime()	返回文件的上次改变时间。	3
filegroup()	返回文件的组 ID。	3
fileinode()	返回文件的 inode 编号。	3
filemtime()	返回文件的上次修改时间。	3
fileowner()	文件的 user ID （所有者）。	3
fileperms()	返回文件的权限。	3
filesize()	返回文件大小。	3
filetype()	返回文件类型。	3
flock()	锁定或释放文件。	3
fnmatch()	根据指定的模式来匹配文件名或字符串。	4
fopen()	打开一个文件或 URL。	3
fpassthru()	从打开的文件中读数据，直到 EOF，并向输出缓冲写结果。	3
fputcsv()	将行格式化为 CSV 并写入一个打开的文件中。	5
fputs()	fwrite() 的别名。	3
fread()	读取打开的文件。	3
fscanf()	根据指定的格式对输入进行解析。	4
fseek()	在打开的文件中定位。	3
fstat()	返回关于一个打开的文件的信息。	4
ftell()	返回文件指针的读/写位置	3
ftruncate()	将文件截断到指定的长度。	4
fwrite()	写入文件。	3

函数	描述	PHP
glob()	返回一个包含匹配指定模式的文件名/目录的数组。	4
is_dir()	判断指定的文件名是否是一个目录。	3
is_executable()	判断文件是否可执行。	3
is_file()	判断指定文件是否为常规的文件。	3
is_link()	判断指定的文件是否是连接。	3
is_readable()	判断文件是否可读。	3
is_uploaded_file()	判断文件是否是通过 HTTP POST 上传的。	3
is_writable()	判断文件是否可写。	4
is_writeable()	is_writable() 的别名。	3
link()	创建一个硬连接。	3
linkinfo()	返回有关一个硬连接的信息。	3
lstat()	返回关于文件或符号连接的信息。	3
mkdir()	创建目录。	3
move_uploaded_file()	将上传的文件移动到新位置。	4
parse_ini_file()	解析一个配置文件。	4
pathinfo()	返回关于文件路径的信息。	4
pclose()	关闭有 popen() 打开的进程。	3
popen()	打开一个进程。	3
readfile()	读取一个文件，并输出到输出缓冲。	3
readlink()	返回符号连接的目标。	3
realpath()	返回绝对路径名。	4
rename()	重名名文件或目录。	3
rewind()	倒回文件指针的位置。	3
rmdir()	删除空的目录。	3
set_file_buffer()	设置已打开文件的缓冲大小。	3
stat()	返回关于文件的信息。	3
symlink()	创建符号连接。	3
tempnam()	创建唯一的临时文件。	3
tmpfile()	建立临时文件。	3
touch()	设置文件的访问和修改时间。	3
umask()	改变文件的文件权限。	3

函数	描述	PHP
unlink()	删除文件。	3

32.12 PHP Filesystem Constants

表 32.4: PHP Filesystem 常量

常量	描述	PHP
GLOB_BRACE		
GLOB_ONLYDIR		
GLOB_MARK		
GLOB_NOSORT		
GLOB_NOCHECK		
GLOB_NOESCAPE		
PATHINFO_DIRNAME		
PATHINFO_BASENAME		
PATHINFO_EXTENSION		
FILE_USE_INCLUDE_PATH		
FILE_APPEND		
FILE_IGNORE_NEW_LINES		
FILE_SKIP_EMPTY_LINES		

Chapter 33

PHP Cookie

cookie 是一种在远程浏览器端储存数据并以此来跟踪和识别用户的机制。

cookie 本身只是服务器存储在用户计算机中的小文件，每次相同的计算机通过浏览器请求页面时，都会同时发送 cookie，这样就可以使用 cookie 来识别用户。

PHP 透明地支持 HTTP cookie，能够通过 setcookie() 或 setrawcookie() 创建 cookie 并通过 $_COOKIE 取回 cookie 的值。

cookie 是 HTTP 标头的一部分，因此 setcookie() 函数必须在其它信息被输出到浏览器前调用，这就和对 header() 函数的限制类似。

用户可以使用输出缓冲函数来延迟脚本的输出，直到按需要设置好了所有的 cookie 或者其它 HTTP 标头。

如果 php.ini 中的配置项 variables_order 中包括“C”，那么任何从客户端发送的 cookie 都会被自动包括进 $_COOKIE 自动全局数组。如果希望对一个 cookie 变量设置多个值，则需要在 cookie 的名称后加 [] 符号。

- 在打开 register_globals 配置选项时，可以从 cookie 建立普通的 PHP 变量。
- 在打开 track_vars 配置选项时，系统还会设定 $HTTP_COOKIE_VARS。

33.1 setcookie()

setcookie() 函数用于设置 cookie。使用时，setcookie() 函数必须位于 <html> 标签之前。

```
setcookie(name, value, expire, path, domain);
```

在下面的例子中，我们将创建名为”user” 的 cookie，把为它赋值”Alex Porter”，同时也规定了此 cookie 在一小时后过期：

```
<?php
setcookie("user", "Alex Porter", time()+3600);
```

```
?>
<!DOCTYPE html>
<html>
<body>

</body>
</html>
```

在发送 cookie 时，cookie 的值会自动进行 URL 编码，在取回时进行自动解码（为防止 URL 编码，要使用 setrawcookie() 取而代之）。

33.2 PHP $_COOKIE

PHP 的 $_COOKIE 变量用于取回 cookie 的值。在下面的例子中取回了名为"user" 的 cookie 的值，并把它显示在了页面上：

```
<?php
// Print a cookie
echo $_COOKIE["user"];

// A way to view all cookies
print_r($_COOKIE);
?>
```

使用 isset() 函数来确认是否已设置了 cookie，示例如下：

```
<!DOCTYPE html>
<html>
<body>

<?php
if (isset($_COOKIE["user"]))
  echo "Welcome " . $_COOKIE["user"] . "!<br />";
else
  echo "Welcome guest!<br />";
?>

</body>
</html>
```

当删除 cookie 时，应当使过期日期变更为过去的时间点。

```
<?php
// set the expiration date to one hour ago
```

```
setcookie("user", "", time()-3600);
?>
```

如果应用程序涉及不支持 cookie 的浏览器，那么开发者就不得不采取其他方法在应用程序中从一张页面向另一张页面传递信息。一种方式是从表单传递数据，下面的表单在用户单击提交按钮时向”welcome.php” 提交了用户输入：

```
<!DOCTYPE html>
<html>
<head>
  <title>PHP Example</title>
</head>
<body>

<form action="welcome.php" method="post">
Name: <input type="text" name="name" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>

</body>
</html>
```

Chapter 34

PHP Session

当用户在本地运行一个应用程序时，可能会执行打开、更改和关闭应用程序的操作，这个过程很像一次会话，而且在这个过程中计算机清楚当前用户是谁，而且它知道我们何时启动应用程序，并在何时终止。

在因特网上运行一个应用程序时会产生一个问题，即服务器不知道当前用户是谁以及用户做什么，原因出在 HTTP 地址不能维持状态（stateless）。

PHP 会话（session）机制可以用来解决上述问题，并且保持用户连续访问 Web 应用时的相关数据，从而有助于创建高度定制化的程序。

Session 本身是一个用户在一段时间内对某一个站点的一次访问，通过使用 Session 在服务器上存储用户信息（比如用户名称、购买商品等）实现随后继续访问用户会话数据。

通常情况下，会话信息都是临时的，在用户离开网站后就将被删除，因此往往需要把数据存储在数据库中来永久储存会话信息。

Session 保存的变量只能供一个用户使用，每一个访客都有自己的 Session 对象变量，因此 Session 对象具有唯一性。

对于一个 Web 应用程序而言，所有用户访问到的 Application 对象的内容是完全一样的，但是不同用户会话访问到的 Session 对象的内容则各不相同，而且存储于 Session 对象中的变量持有单一用户的信息，并且对于一个应用程序中的所有页面都是可用的。

PHP 为每位用户创建一个唯一的 cookie，cookie 被传送至客户端，它含有可识别用户的信息，这种接口被称作 Session 对象。PHP Session[1] 变量用于存储有关用户会话的信息，或更改用户会话的设置。Session 变量保存的信息是单一用户的，并且可供应用程序中的所有页面使用。

Session 的工作机制是为每个访问者创建一个唯一的 id (UID)，并基于这个 UID 来存储变量。UID 存储在 cookie 中，亦或通过 URL 进行传导，因此用户在应用程序的页面切换时，

[1] Session 对象在.NET 中对应 HttpSessionState 类，表示“会话状态”，可以保存与当前用户会话相关的信息。

Session 对象的变量不会被清除。这样，Session 就可以存储从一个用户开始访问某个特定的 PHP 页面起，到用户离开为止，特定的用户会话相关的信息。

存储于 Session 对象中的信息通常是 name、id 以及参数。服务器会为每个新的用户创建一个新的 Session，并在 Session 到期时撤销掉这个 Session 对象。

Session 开始于：

- 当某个新用户请求了一个 PHP 文件，并且 PHP 文件中引用了 `sesstion_start()` 函数时；
- 当某个值存储在 Session 变量中时；

使用 session 时主要的问题是它们该在何时结束。我们不会知道用户最近的请求是否是最后的请求。因此我们不清楚该让 session“存活”多久。为某个空闲的 session 等待太久会耗尽服务器的资源。然而假如 session 被过早地删除，那么用户就不得不一遍又一遍地重新开始，这是因为服务器已经删除了所有的信息。寻找合适的超时间隔时间是很困难的，因此如果正在使用 session 变量，尽量不要在其中存储大量的数据。

34.1 PHP Session Lifecycle

One of the most vast misconceptions in the PHP world is how sessions really do work[?]. I'm fairly confident most of us know how to start sessions, terminate sessions, regenerate session IDs and easily pass data from 1 page to another. But do you really know how they work inside out?

Storing crucial information in a session is what sessions are all about. They strive on being able to provide the information without leaving it open to tampering or interception. Giving a session some important information is like giving a Jack Russell a bone - it's not letting go of it anytime soon and if any other sod tries to take the bone.

Anything can be unsafe in PHP if the programming is not up to par. Generally speaking, however, sessions are 1 section of the PHP language that is difficult to mess up.

Sessions are conveniently stored server-side. There is no exception to this rule. Many people become confused because sessions use cookies, and cookies, rightfully so, are stored client-side. Sessions, however, are not cookies in the truest form. Cookies are just 1 of the methods of delivering the unique session ID to retrieve the session data.

Remember the times when you used to crack open the cereal boxes to retrieve the free plastic toy from inside? Well, sessions are not quite as exciting as that. In fact, sessions are pretty uneventful on the inside.

```
<?php
  session_start();
  $_SESSION['myWebsite'] = 'http://www.theqiong.com/';
  $_SESSION['mySessionId'] = session_id();
```

```
?>
```

Request Header:

```
GET /test.php HTTP/1.1
Host: localhost
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/31.0.1650.57 Safari/537.36
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8,ja;q=0.6,zh-CN;q=0.4,zh-TW;q=0.2
```

Request Header:

```
HTTP/1.1 200 OK
Date: Thu, 28 Nov 2013 14:13:31 GMT
Server: Apache/2.4.6 (Fedora) PHP/5.5.5 SVN/1.7.13
X-Powered-By: PHP/5.5.5
Set-Cookie: PHPSESSID=h589n8vacg3jlrjm82ia11mff3; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

As simple as it may be, it does the job. It creates us a session file which contains our session information. The file is stored as the following filename which is made up of 2 parts: it has prefixed us the word sess_ which is then followed by our unique session ID.

`sess_h589n8vacg3jlrjm82ia11mff3`

If you open up this file it contains the following information which is simply my array which has been serialized (see serialize for further information) and stored.

`myWebsites:24:"http://www.theqiong.com/";myId|s:26:"h589n8vacg3jlrjm82ia11mff3";|`

This is nothing more than my array broken down into a string. When I execute my file again the session ID will be used to retrieve the above data from the relevant session file and unserialized (see unserialize for further information).

That's all there really is to a session file. Remember that the session file is stored server-side so there is absolutely no need to transfer any information you do not publish in the HTML over the insecure medium we like to call the Internet.

34.2 session_start()

在用户把用户信息存储到 PHP session 中之前，首先必须通过 session_start() 函数启动会话，而且 session_start() 函数必须位于 <html> 标签之前。

```
<?php session_start(); ?>
<!DOCTYPE html>
<html>
<head>
  <title>PHP Session Example</title>
</head>
<body>

</body>
</html>
```

上面的代码会向服务器注册用户的会话，以便可以开始保存用户信息，同时会为用户会话分配一个 UID。一旦值被存入 session 变量，它就能被 PHP 应用程序中的任何页面使用，Session 对象最大的优点是可在其中存储变量，以供后续的网页读取，其应用范围是很广的。

34.3 Session ID

Cookies are the most frequently used due to the extra security they add (not much, but just enough to be the favoured method). The other possible methods are GET (which is sometimes used) and POST.

Unlike GET and POST which can be used in such attacks as CSRF and passed around to potential victims with far too much ease, cookies are just that little more difficult to install on a victim's computer.

You could quite easily send a link to someone and have him either hijack a session on my behalf, or the most common reason would be to fixate session ID (see the following post for more information about session fixation).

Well, there is an important setting that is set to 1 by default (and should always be set to 1 unless you wish to also support users who do not have cookies enabled - which is not recommended) that prevents the session IDs from being transferred via a GET and to use only cookies - the safest delivery method. That setting is:

```
session.use_only_cookies
```

From the earlier example let's take a look at the cookie that has been created on local computer:

`h589n8vacg3jlrjm82ia11mff3`

The cookie's name is `PHPSESSID` which is the default name given to a session cookie that is stored client-side. The number you see above is the session ID which will stay with me for the lifetime of my session or until it is regenerated for security purposes.

Upon loading the website where my session ID was created, the cookie is sent to the website and parsed by PHP. The session ID is correctly linked to the session file stored server-side and is then unserialized and placed into the pre-populated PHP array, `$_SESSION`.

Once issued the `session_start()` command to begin session, then we're able to access all the information - even crucial and highly sensitive information, from the `$_SESSION` array.

Although you may store highly insensitive information in sessions, be aware that displaying it on your website will mean that crucial information is being sent over the pipes of the Internet. Without using SSL this would be a very insecure method of delivery which would be crying out to be intercepted by Harry the hacker.

34.4 PHP $_SESSION

存储和取回 session 变量的正确方法是使用 PHP $_SESSION 变量：

```
<?php
session_start();
// store session data
$_SESSION['views']=1;
?>
<!DOCTYPE html>
<html>
<head>
  <title>PHP Session Example</title>
</head>
<body>

<?php
//retrieve session data
echo "Pageviews=". $_SESSION['views'];
?>

</body>
</html>
```

在下面的例子中，我们创建了一个简单的 page-view 计数器。isset() 函数检测是否已设置 “views” 变量。如果已设置 “views” 变量，我们累加计数器。如果 “views” 不存在，则我们创建 “views” 变量，并把它设置为 1：

```
<?php
session_start();

if(isset($_SESSION['views']))
  $_SESSION['views']=$_SESSION['views']+1;

else
  $_SESSION['views']=1;
echo "Views=". $_SESSION['views'];
?>
```

`$_SESSION` 变量与 `$_POST`、`$_GET` 变量相比，`$_SESSION` 变量要在 `session_start()` 函数执行后才初始化，而后两者是在 PHP 页面请求时就已经初始化了。

Assume that banking system has absolutely no extra security implemented, and some customer have logged in and been issued a cookie containing their session ID, banking system also erroneously accepts the GET method of delivery though to ensure everybody can use their system.

Harry decides to go down the route of session fixation and thus sends you the following crafty link:

`http://www.myOnlineBank.com/index.php?sess_id=5aff2`

If you click on it and login. Harry, already knowing your session ID is 5aff2, clicks on the link himself and is able to withdraw your entire life savings of 67 pence.

Nonetheless, the browser name, version, language, etc. are all sent by the end user's browser during the HTTP call. This means that this data may not even be sent at all. It is optional. As long as you know it is optional you can check for if the intended array even exists before using it. This is important because a blank string will always produce the same MD5 or SHA1 (etcetera...) hash. A null string MD5'd will always be d41d8cd98f00b204e9800998ecf8427e no matter how many times you use MD5. Whilst SHA1 will be different to the MD5 string, it will still be the same for every single time you hash the null string using SHA1.

Now that you fully understand the following:

`[Session File] (Server) -> [Cookie File] (Client)`

A session file can simply be opened by anybody who has access to file system. For instance, sessions are stored in the following directory as plain files:

`C:\wamp\www`

Anybody who has access to that directory can read session files and read the session IDs from the filenames. This is why shared hosts can be bad. Typically everybody on a shared hosts shares the same temp folder. This is the most common place where session files are stored.

Luckily the directory can be moved elsewhere or you can even use a table inside a database to store all your session data. The latter is best saved for another article to keep this 1 short and sweet.

The following setting can be changed to alter the destination of session files:

`session.save_path`

34.5 unset()

如果希望删除某些 session 数据，可以使用 unset() 或 session_destroy() 函数，其中 unset() 函数用于释放指定的 session 变量：

```
<?php
unset($_SESSION['views']);
?>
```

34.6 session_distroy()

通过 session_destroy() 函数可以彻底终结 session，session_destroy() 将重置 session，因此将失去所有已存储的 session 数据。

```
<?php
session_destroy();
?>
```

Chapter 35

PHP Forms

表单是 Web 应用程序和用户交互时重要的手段，Web 设计中常见的表单示例如下：

```
<!DOCTYPE html>
<html>
<head>
<title>PHP POST Example</title>
</head>
<body>

<form action="welcome.php" method="post">
Name: <input type="text" name="name" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
</body>
</html>
```

上面的 HTML 页面实例包含了两个输入框和一个提交按钮。当用户填写了该表单并单击了提交按钮后，页面 welcome.php 将被调用。表单的数据会被送往”welcome.php” 这个文件，而 “welcome.php” 文件类似这样：

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome</title>
</head>
<body>

Welcome <?php echo $_POST["name"]; ?>. You are <?php echo $_POST["age"]; ?> years old.
</body>
</html>
```

PHP 的 $_GET 变量和 $_POST 变量用于检索表单中的值，比如用户输入等。

- $_GET 变量用于收集来自 method="get" 的表单中的值。
- $_POST变量用于收集来自 method="post" 的表单中的值。

更严格的 PHP 脚本如下：

```
Welcome <?php echo htmlspecialchars($_POST['name']); ?>.<br />
You are <?php echo (int)$_POST["age"]; ?> years old.
```

htmlspecialchars() 使得 HTML 之中的特殊字符被正确的编码，从而不会被使用者在页面注入 HTML 标签或者 Javascript 代码。

我们明确知道 age 字段是一个数值，因此将它转换为一个整形值 (integer) 来自动的消除任何不必要的字符。也可以使用 PHP 的 filter 扩展来自动完成该工作。

除了 htmlspecialchars() 和 (int) 部分，这段程序做什么用显而易见。

PHP 一个很有用的特点体现在它处理 PHP 表单的方式。当一个表单提交给 PHP 脚本时，表单的任何元素都在 PHP 脚本中自动生效，因此 PHP 将自动设置 $_POST['name'] 和 $_POST['age'] 变量。在这之前我们使用了超全局变量 $_SERVER，现在引入超全局变量是 $_POST，它包含了所有的 POST 数据。

如果表单提交数据的方法（method）使用的是 GET 方法，那么表单中的信息将被储存到超全局变量 $_GET 中。

- GET[1]

  通过 HTTP GET 机制，把参数数据队列加到提交表单的 action 属性所指的 URL 中，值和表单内各个字段一一对应，在 URL 中可以看到，因此 GET 安全性非常低，建议用 POST 方式传输机密信息[2]。

  通过 HTTP GET 机制传送的数据量较小，不能大于 2KB，因此 GET 执行效率比 POST 方法好。

- POST[3]

  通过 HTTP POST 机制，将表单内各个字段与其内容放置在 HTML header 内一起传送到 action 属性所指的 URL 地址，用户是看不到这个过程的，因此 POST 安全性较高。

  通过 HTTP POST 机制传送的数据量较大，一般被默认为不受限制。但理论上，IIS4 中最大量为 80KB，IIS5 中为 100KB。

如果并不关心请求数据的来源，也可以用超全局变量 $_REQUEST，它包含了所有 GET、POST、COOKIE 和 FILE 的数据。

通常，PHP 不会改变传递给脚本中的变量名，但是应该注意到点（句号）不是 PHP 变量名中的合法字符，因此变量名中的点和空格会被转换成下划线。例如 <input name="a.b"

[1]对于 GET 方式，服务器端用 Request.QueryString 获取变量的值。

[2]具体来说，在做数据查询时，建议用 GET 方式，而在做数据添加、修改或删除时，建议用 POST 方式。

[3]对于 POST 方式，服务器端用 Request.Form 获取提交的数据。

/> 变成了 $_REQUEST["a_b"]。

```
<?php
$varname.ext; /* 非法变量名 */
?>
```

这时，解析器看到是一个名为 $varname 的变量，后面跟着一个字符串连接运算符，后面跟着一个裸字符串（即没有加引号的字符串，且不匹配任何已知的健名或保留字）'ext'。很明显这不是想要的结果，因此出于此原因，要注意 PHP 将会自动将变量名中的点替换成下划线。

当提交表单时，可以用一幅图像代替标准的提交按钮，用类似这样的标记：

```
<input type="image" src="image.gif" name="sub" />
```

当用户点击到图像中的某处时，相应的表单会被传送到服务器，并加上两个变量 sub_x 和 sub_y。它们包含了用户点击图像的坐标。有经验的用户可能会注意到被浏览器发送的实际变量名包含的是一个点而不是下划线（即 sub.x 和 sub.y），但 PHP 自动将点转换成了下划线。

根据特定的设置和个人的喜好，有很多种方法访问 HTML 表单中的数据。例如：

```
<?php
// 自 PHP 4.1.0 起可用
   echo $_POST['username'];
   echo $_REQUEST['username'];

   import_request_variables('p', 'p_');
   echo $p_username;

// 自 PHP 5.0.0 起，这些长格式的预定义变量
// 可用 register_long_arrays 指令关闭。

   echo $HTTP_POST_VARS['username'];

// 如果 PHP 指令 register_globals = on 时可用。不过自
// PHP 4.2.0 起默认值为 register_globals = off。
// 不提倡使用/依赖此种方法。

   echo $username;
?>
```

使用 GET 表单也类似，只不过要用适当的 GET 预定义变量。在 PHP 4.2.0 之前 register_globals 的默认值是 on。PHP 社区不鼓励依赖此指令，建议在编码时假定其为 off。

GET 也适用于 `QUERY_STRING`(URL 中在“?”之后的信息)。因此，举例说，http://www.example.com/test.php?i 包含有可用 `$_GET['id']` 来访问的 GET 数据。

另外，magic_quotes_gpc 配置指令影响到 GET，POST 和 COOKIE 的值。如果打开，值 `(It's "PHP!")` 会自动转换成 `(It\'s \"PHP!\")`。十多年前对数据库的插入需要如此转义，如今已经过时了，应该关闭。

PHP 也懂得表单变量上下文中的数组，例如可以将相关的变量编成组，或者用此特性从多选输入框中取得值。例如，将一个表单 POST 给自己并在提交时显示数据：

```
<?php
if (isset($_POST['action']) && $_POST['action'] == 'submitted') {
   echo '<pre>';

   print_r($_POST);
   echo '<a href="'. $_SERVER['PHP_SELF'] .'">Please try again</a>';

   echo '</pre>';
} else {
?>
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
   Name: <input type="text" name="personal[name]"><br />
   Email: <input type="text" name="personal[email]"><br />
   Beer: <br>
   <select multiple name="beer[]">
      <option value="warthog">Warthog</option>
      <option value="guinness">Guinness</option>
      <option value="stuttgarter">Stuttgarter Schwabenbr</option>
   </select><br />
   <input type="hidden" name="action" value="submitted" />
   <input type="submit" name="submit" value="submit me!" />
</form>
<?php
}
?>
```

应该在任何可能的时候对用户输入进行验证，客户端验证的优势在于速度更快，从而可以减轻服务器的负载。

不过，任何流量很高以至于不得不担心服务器资源的站点，也有必要担心站点的安全性。如果表单访问的是数据库，就非常有必要采用服务器端的验证。

在服务器验证表单的一种好的方式是，把表单传给它自己，而不是跳转到不同的页面。

这样用户就可以在同一张表单页面得到错误信息，也就更容易发现错误了。

35.1 XForms

XForms 定义了一种传统 web 表单的变种，可以用于更多的平台和浏览器（包括非传统的媒体（例如 PDF 文档)），PHP 支持处理 XForms 的输入，尽管用户可能更喜欢使用长久以来支持良好的 HTML 表单。

XFroms 的第一个关键区别是表单怎样发送到客户端的信息。例如，下面的 XForms 表单显示一个文本输入框（命名为 q）和一个提交按钮，当用户点击提交按钮后，表单将被发送到 action 所指示的页面。

Example 29 XForms 搜索表单

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
       xmlns="http://www.w3.org/2002/xforms">
<h:head>
 <h:title>Search</h:title>
 <model>
  <submission action="http://example.com/search"
            method="post" xml:id="s"/>
 </model>
</h:head>
<h:body>
 <h:p>
  <input ref="q"><label>Find</label></input>
  <submit submission="s"><label>Go</label></submit>
 </h:p>
</h:body>
</h:html>
```

- 在普通的 HTML 表单中，数据发送格式是 application/x-www-form-urlencoded；
- 在 XForms 表单中，数据发送格式是 XML。

XForms 传送回服务器后，$HTTP_RAW_POST_DATA 中包含了由浏览器产生的 XML 文档，可以将其传递给相应的 XSLT 引擎或者文档解析器。

如果需要让数据保存到传统的 $_POST 变量中，可以将 method 属性改成 urlencoded-post，并且指示客户端浏览器将其以 application/x-www-form-urlencoded 格式发送给服务器。

Example 30 使用 XForm 来产生 $_POST

```
<h:html xmlns:h="http://www.w3.org/1999/xhtml"
```

```
      xmlns="http://www.w3.org/2002/xforms">
<h:head>
 <h:title>Search</h:title>
 <model>
  <submission action="http://example.com/search"
            method="urlencoded-post" xml:id="s"/>
 </model>
</h:head>
<h:body>
 <h:p>
  <input ref="q"><label>Find</label></input>
  <submit submission="s"><label>Go</label></submit>
 </h:p>
</h:body>
</h:html>
```

Chapter 36

PHP $_GET

$_GET 变量是一个数组，内容是由 HTTP GET 方法[1]发送的变量名称和值。

$_GET 变量用于收集来自 method="get" 的表单中的值。从带有 GET 方法的表单发送的信息，对任何人都是可见的（会显示在浏览器的地址栏），并且对发送的信息量也有限制（最多 100 个字符）。

```
<form action="welcome.php" method="get">
Name: <input type="text" name="name" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
```

当用户点击提交按钮时，发送的 URL 会类似这样：

```
http://www.domain.com.cn/welcome.php?name=Jim&age=30
```

"welcome.php" 文件通过$_GET 变量来获取表单数据时，表单域的名称会自动成为$_GET数组中的 ID 键：

```
Welcome <?php echo $_GET["name"]; ?>.<br />
You are <?php echo $_GET["age"]; ?> years old!
```

在使用$_GET变量时，所有的变量名和值都会显示在 URL 中。所以在发送密码或其他敏感信息时，不应该使用这个方法。不过，正因为变量显示在 URL 中，因此可以在收藏夹中收藏该页面。在某些情况下，这是很有用的。

[1]HTTP GET 方法不适合大型的变量值，不能超过 100 个字符。

Chapter 37

PHP $_POST

$_POST变量是一个数组，内容是由 HTTP POST 方法发送的变量名称和值。

$_POST变量用于收集来自 method="post" 的表单中的值。从带有 POST 方法的表单发送的信息，对任何人都是不可见的（不会显示在浏览器的地址栏），并且对发送信息的量也没有限制。

```
<form action="welcome.php" method="post">
Enter your name: <input type="text" name="name" />
Enter your age: <input type="text" name="age" />
<input type="submit" />
</form>
```

当用户点击提交按钮，URL 不会含有任何表单数据，看上去类似这样：

```
http://www.domain.com.cn/welcome.php
```

"welcome.php" 文件现在可以通过$_POST变量来获取表单数据时，表单域的名称会自动成为$_POST数组中的 ID 键。

```
Welcome <?php echo $_POST["name"]; ?>.<br />
You are <?php echo $_POST["age"]; ?> years old!
```

使用$_POST，通过 HTTP POST 发送的变量不会显示在 URL 中，而且变量没有长度限制。不过，由于变量不显示在 URL 中，所以无法把页面加入书签。

Chapter 38

PHP $_REQUEST

PHP 的$_REQUEST变量包含了$_GET, $_POST以及$_COOKIE的内容。

PHP 的$_REQUEST变量可用来取得通过GET和POST等方法发送的表单数据的结果。

```
Welcome <?php echo $_REQUEST["name"]; ?>.<br />
You are <?php echo $_REQUEST["age"]; ?> years old!
```

Chapter 39

PHP Image

除了处理字符串等文本数据之外，PHP 也可以使用 GD 库等来处理图像（例如文字按钮、计数器数字、股票走势图形、数据拼图等静态或动态图形)。

基本的图像文件操作包括对点、线和基本几何图形的实现，以及书写动态文字、使用调色板、填充图形区域等。

PHP 在使用图形库来创建图像时，首先在内存中建立图像，然后输出并下载到用户的浏览器中。

1. 创建画布来开始绘图设计，画布实际上就是在内存中开辟的一块用于存储图像信息的临时区域;
2. 设置图像颜色来开始使用图像处理函数进行具体绘图;
3. 图像绘制完成后转交给 Web 服务器，并发送（或下载）到浏览器来进行显示;
4. 图像发送完成后就会被及时清理内存中的虚拟“画布”。

39.1 Pixel

像素（pixel）是构成数字图像的最小单位，若干个像素点以矩阵的方式排列后就形成了图像。

39.2 Palette

调色板（palette）是一个包含颜色值的数组，像素的颜色由它们在调色板中的位置确定。

调色板中的每一项都是由 3 个独立的颜色值（红、绿、蓝）混合而成，它们的范围都可以从 0 ～ 255.

0 代表不显示颜色，255 代表颜色的浓度最深，因此每个像素最多可以有 16777216（即 256x256x256）种可能的颜色。

图像以像素和调色板作为数据，并按照一定的算法存储为文件，因此不同的图像文件格式有不同的存储算法。

通常情况下，GIF、JPEG 和 PNG 等图像文件格式都是压缩的，可以减小文件的体积。

39.3 Bit

颜色也可以理解为由数字（位，Bit）表示的，并且颜色数 $=2^n$，其中 n 就是所占的位数。

- 高彩色指 16 位显示模式，支持 65536（64K）种颜色；

$$2^{16} = 65536$$

- 真彩色指 24 位显示模式，支持 1677 万（16M）种颜色。

$$2^{24} = 16777216$$

39.4 Index

图像指数可以评价图像整体的感观和效果，并通过对比度、亮度、色调和饱和度等参数进行量化表达。

- 对比度是画面最亮的部分和最暗的部分的比值，其值越大则明暗对比越强；
- 亮度是画面总体的明亮程度，其值越大则画面越明亮；
- 色调是色偏，可以分为冷色调和暖色调等；
- 饱和度是色调的光度或暗度以及色调的纯度。

饱和度表示图像的彩色深浅度或鲜艳度，取决于彩色种的白色光含量。白色光含量越高，则彩色光含量越低，色彩饱和度就越低。

39.5 Transparency

透明度控制背景是否可以透过图像，以及以多大的恶透明度透过图像。

PNG 等支持 Alpha 通道，这样每一个 ie 像素都有一个额外的值来表示其透明度，其他图像只是简单地指定调色板中的选项值来指定透明度。

39.6 GD

GD 库本身是一个 PHP 动态链接扩展库，可以使 PHP 处理 GIF、JPEG、PNG、WBMP、XBM 等格式的图像文件操作，并且支持额外的字体库（例如 FreeType 和 Type 1 字体等）。

在 Windows 自动捆绑了 GD 库，但是默认不开启，因此需要在 php.ini 中通过“extension=php_gd2.dll”选项来手动打开对 GD 扩展库的支持。

在 UNIX/Linux 中需要在编译 PHP 时使用“`--with-gd`”选项打开，而且需要安装相关的图片格式和字体。

默认情况下，GD 库的功能有限，需要配合其他的函数库进行使用。

39.6.1 gd_info()

gd_info() 取得当前安装的 GD 库的信息，并返回一个关联数组描述了安装的 GD 库的版本和性能。

39.7 Canvas

39.7.1 imageCreate)

39.7.2 imageCreateTrueColor()

39.8 Color

39.8.1 imageColorAllocate()

39.9 Image

PHP 可以直接生成 GIF、JPEG、PNG 和 WBMP 四种图像格式的文件，也可以从不同的源（例如本地或网络图像文件）新建图像。

在直接输出图像到浏览器之前，需要使用 header() 发送标头信息来设置正确的 MIME 文件类型。

- Content-type: image/gif
- Content-type: image/jpeg
- Content-type: image/png
- Content-type: image/vnd.wap.wbmp

在保存图像到服务器上时，不需要使用 header() 发送标头信息。

39.9.1 imageGIF()

39.9.2 imageJPEG()

39.9.3 imagePNG()

39.9.4 imageWBMP()

39.9.5 imageCreateFromGIF()

39.9.6 imageCreateFromJPEG()

39.9.7 imageCreateFromPNG()

39.9.8 imageCreateFromWBMP()

39.9.9 imageCreateFromString()

39.9.10 imageDestroy()

39.10 Graph

在使用图形库进行点、线和面的绘制时，图形在画布中的位置通过坐标进行确定。

- 坐标原点在画布左上角的起始位置，以像素大小为单位。
- X 轴正方向向右延伸;
- Y 轴正方向向下延伸。

39.10.1 imageSetPixel()

39.10.2 imageLine()

39.10.3 imageDashedLine()

```
<?php
  //创建图像
  $img = imageCreate (300, 200);
  $background = ImageColorAllocate($img, 255, 255, 255);   //背景设为白色
  $black = ImageColorAllocate ($img, 0, 0, 0);     //设定黑色
  $white = $background;            //设定白色

  //发送标头信息
  header("Content-type: image/png");

  //绘制矩形
  imageRectangle($img, 30, 30, 180, 120, $black);
  imageRectangle($img, 60, 60, 210, 150, $black);

  //填充颜色
  imageFill ($img, 100, 100,$black);

  //输出文字
  imageString ($img, 5, 75, 81, "Hello, PHP", $white);

  //输出PNG图像
  imagePNG($img);

  //销毁图像
  imageDestroy($img);
?>
```

39.10.4 imageRectangle()

39.10.5 imagePolygon()

39.10.6 imageArc()

39.10.7 imageSetThickness()

```
<?php
  //定义PNG图片的高和宽
  $widht =501;
  $height=201;

  //创建图片
  $image=imageCreate($widht, $height);

  //定义颜色
  $colorWhite=imageColorAllocate($image, 255, 255, 255);
  $colorGrey=imageColorAllocate($image, 192, 192, 192);
  $colorBlue=imageColorAllocate($image, 0, 0, 255);
  $colorRed =imageColorAllocate($image, 255, 0, 0);

  //创建包围在图片周围的线
  for ($i=0; $i<=30; $i++)
  {
    imageline($image, 0, $i*20, 500, $i*20, $colorGrey);
    imageline($image, $i*20, 0, $i*20, 200, $colorGrey);
  }

  //绘制多边形
  $graphValues=array(1,142,45,190,80,181,120,105,23,13,150,55);
  imagePolygon($image, $graphValues, 6, $colorBlue);

  //绘制弧形
  imageArc($image, 250, 100, 200, 150, 40, 320, $colorRed);

  //绘制圆形
  imageEllipse($image, 400, 100, 179, 179, $colorBlue);

  //设定图形的线宽, 并绘制矩形
  imageSetThickness($image, 3);
  imageRectangle($image, 250, 52, 400, 146, $colorRed);
```

```
  //输出PNG图形
  header("Content-type: image/png");
  imagePNG($image);

  //销毁资源
  imageDestroy($image);
?>
```

39.10.8 getImageSize()

在对图像文件进行操作之前，可以使用 getImageSize() 函数获得图片的尺寸等相关信息。

39.10.9 imageCopy()

39.10.10 imageCopyMerge()

39.10.11 imageCopyMergeGray()

39.10.12 imageCopyResampled()

39.10.13 imageCopyResized()

39.10.14 imageRotate()

39.11 Fill

39.11.1 imageFill()

39.11.2 imageFillToBorder()

39.11.3 imageFilledRectangle()

39.11.4 imageFilledPolygon()

39.11.5 imageFilledArc()

39.11.6 imageFilledEllipse()

39.12 String

39.12.1 imageString()

39.12.2 imageStringUp()

39.12.3 imageChar()

39.12.4 imageCharUp()

39.12.5 imageLoadFont()

除了使用内置的字体之外，PHP 还可以使用 imageLoadFont() 函数加载用户自定义字体文件，并返回对应字体的标识符。

字体文件的格式是二进制的，并且和平台有关，因此不同类型的 CPU 的机器生成的字体是不同的。

39.12.6 imageTtfText()

TrueType 是一种可以缩放的，与设备无关的字体，并且可以按轮廓存储 TrueType 字体，因此它们可以任意调整高度，打印 TrueType 字体时可以与屏幕上显示的完全相同。

39.12.7 imageTtfbBox()

39.13 Filter

PHP 提供的图像过滤器函数可以对图形进行修饰，这些内置的过滤器类似于 Photoshop 中的“滤镜”功能。

39.13.1 imageFilter()

下面的说明如何对图像进行灰度处理。

```
<?php
  //打开图像
  $im = imageCreateFromPNG('dave.png');

  if ($im && imageFilter($im, IMG_FILTER_GRAYSCALE))
```

```
  {
       echo '灰度处理成功! ';
     imagePNG($im, 'dave.png');
  } else {
       echo '灰度处理失败! ';
  }

  //消除内存图像
  imageDestroy($im);
?>
```

下面的说明如何对图像进行模糊处理。

```
<?php
  //打开图像
  $im = imageCreateFromPNG('sean.png');

  if ($im && imageFilter($im, IMG_FILTER_GAUSSIAN_BLUR))
  {
     echo '亮度改变成功! ';
     imagePNG($im, 'sean.png');
  }else{
     echo '亮度改变失败! ';
  }

  //消除内存图像
  imageDestroy($im);
?>
```

Chapter 40

PHP Mail

绝大多数邮件服务器都支持 SMTP、POP3 和 IMAP 协议，PHP 可以基于邮件协议来实现邮件的收取、发送等服务。

40.1 mail()

PHP 允许从脚本直接发送电子邮件，基本语法如下：

```
mail(to,subject,message,headers,parameters)
```

通过 PHP 发送电子邮件的最简单的方式是发送一封文本 email。在下面的例子中，我们首先声明变量 ($to, $subject, $message, $from, $headers)，然后我们在 mail() 函数中使用这些变量来发送了一封 E-mail：

```
<?php

$to = "someone@example.com";
$subject = "Test mail";
$message = "Hello! This is a simple email message.";
$from = "someonelse@example.com";
$headers = "From: $from";
mail($to,$subject,$message,$headers);
echo "Mail Sent.";

?>
```

表 40.1: PHP Mail 配置选项

名称	默认	描述	可改变
SMTP	“localhost”	Windows 专用：SMTP 服务器的 DNS 名称或 IP 地址。	PHP_INI_ALL
smtp_port	“25”	Windows 专用：SMTP 段口号。自 PHP 4.3 起可用。	PHP_INI_ALL
sendmail_from	NULL	Windows 专用：规定从 PHP 发送的邮件中使用的”from” 地址。	PHP_INI_ALL
sendmail_path	NULL	Unix 系统专用：规定 sendmail 程序的路径（通常 /usr/sbin/sendmail 或 /usr/lib/sendmail）	PHP_INI_SYSTEM

40.2 PHP Mail Functions

邮件函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

表 40.2: PHP Mail 函数

函数	描述	PHP
ezmlm_hash()	计算 EZMLM 邮件列表系统所需的散列值。	3
mail()	允许从脚本中直接发送电子邮件。	3

PHP 需要一个已安装且正在运行的邮件系统，以便使邮件函数可用。所用的程序通过在 php.ini 文件中的配置设置进行定义，也就是说，邮件函数的行为受 php.ini 的影响。

表 40.3: PHP mail() 函数参数

参数	描述
to	必需。规定 email 接收者。
subject[1]	必需。规定 email 的主题。
message	必需。定义要发送的消息。应使用 LF (\n) 来分隔各行。
headers	可选。规定附加的标题，比如 From、Cc 以及 Bcc。 应当使用 CRLF (\r\n) 分隔附加的标题。
parameters	可选。对邮件发送程序规定额外的参数。

[1]注释：该参数不能包含任何新行字符。

40.3 PHP Mail Form

通过 PHP 可以在自己的站点制作一个反馈表单。下面的例子向指定的 e-mail 地址发送了一条文本消息：

```
<!DOCTYPE html>
<html>
<body>

<?php
if (isset($_REQUEST['email']))
//if "email" is filled out, send email
  {
  //send email
  $email = $_REQUEST['email'] ;
  $subject = $_REQUEST['subject'] ;
  $message = $_REQUEST['message'] ;
  mail( "someone@example.com", "Subject: $subject",
  $message, "From: $email" );
  echo "Thank you for using our mail form";
  }
else
//if "email" is not filled out, display the form
  {
  echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";
  }
?>

</body>
</html>
```

1. 首先，检查是否填写了邮件输入框
2. 如果未填写（比如在页面被首次访问时），输出 HTML 表单
3. 如果已填写（在表单被填写后），从表单发送邮件
4. 当点击提交按钮后，重新载入页面，显示邮件发送成功的消息

40.4 PHP Mail Injection

在上述 PHP e-mail 脚本中，存在着一个漏洞，导致未经授权的用户可通过输入表单在邮件头部插入数据。

假如用户在表单中的输入框内加入这些文本，会出现什么情况呢？

```
someone@example.com%0ACc:person2@example.com
%0ABcc:person3@example.com,person3@example.com,
anotherperson4@example.com,person5@example.com
%0ABTo:person6@example.com
```

与往常一样，mail() 函数把上面的文本放入邮件头部，那么现在头部有了额外的 Cc:, Bcc: 以及 To: 字段。当用户点击提交按钮时，这封 e-mail 会被发送到上面所有的地址！

40.5 PHP Mail Validation

防止 e-mail 注入的最好方法是对输入进行验证。

下面的代码与上一节类似，不过已经增加了检测表单中 email 字段的输入验证程序：

```
<!DCOTYPE html>
<html>
<body>
<?php
function spamcheck($field)
  {
  //filter_var() sanitizes the e-mail
  //address using FILTER_SANITIZE_EMAIL
  $field=filter_var($field, FILTER_SANITIZE_EMAIL);

  //filter_var() validates the e-mail
  //address using FILTER_VALIDATE_EMAIL
  if(filter_var($field, FILTER_VALIDATE_EMAIL))
    {
    return TRUE;
    }
  else
    {
    return FALSE;
    }
  }

if (isset($_REQUEST['email']))
```

```
  {//if "email" is filled out, proceed

  //check if the email address is invalid
  $mailcheck = spamcheck($_REQUEST['email']);
  if ($mailcheck==FALSE)
    {
    echo "Invalid input";
    }
  else
    {//send email
    $email = $_REQUEST['email'] ;
    $subject = $_REQUEST['subject'] ;
    $message = $_REQUEST['message'] ;
    mail("someone@example.com", "Subject: $subject",
    $message, "From: $email" );
    echo "Thank you for using our mail form";
    }
  }
else
  {//if "email" is not filled out, display the form
  echo "<form method='post' action='mailform.php'>
  Email: <input name='email' type='text' /><br />
  Subject: <input name='subject' type='text' /><br />
  Message:<br />
  <textarea name='message' rows='15' cols='40'>
  </textarea><br />
  <input type='submit' />
  </form>";
  }
?>

</body>
</html>
```

在上面的代码中，我们使用了 PHP 过滤器来对输入进行验证：

- FILTER_SANITIZE_EMAIL 从字符串中删除电子邮件的非法字符
- FILTER_VALIDATE_EMAIL 验证电子邮件地址

Chapter 41

PHP Filter

几乎所有 Web 应用程序都依赖外部的输入，这些外部数据通常来自用户或其他应用程序（比如 Web 服务等），通常包括：

- 来自表单的输入数据
- Cookies
- 服务器变量
- 数据库查询结果

PHP 过滤器可以用于验证和过滤来自非安全来源的数据，比如用户的输入。设计 PHP 的过滤器扩展的目的是使数据过滤更轻松快捷。

如需过滤变量，可以使用下面的过滤器函数之一：

- filter_var() - 通过一个指定的过滤器来过滤单一的变量
- filter_var_array() - 通过相同的或不同的过滤器来过滤多个变量
- filter_input - 获取一个输入变量，并对它进行过滤
- filter_input_array - 获取多个输入变量，并通过相同的或不同的过滤器对它们进行过滤

在下面的例子中，我们用 filter_var() 函数验证了一个整数：

```
<?php
$int = 123;

if(!filter_var($int, FILTER_VALIDATE_INT))
 {
 echo("Integer is not valid");
 }
else
 {
 echo("Integer is valid");
 }
```

```
?>
```

上面的代码使用了 FILTER_VALIDATE_INT 过滤器来过滤变量。由于这个整数是合法的，因此代码的输出是：Integer is valid。假如尝试使用一个非整数的变量，则输出是：Integer is not valid。

41.1 PHP Filter Functions

filter 函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

表 41.1: PHP Filter 函数

函数	描述	PHP
filter_has_var()	检查是否存在指定输入类型的变量。	5
filter_id()	返回指定过滤器的 ID 号。	5
filter_input()	从脚本外部获取输入，并进行过滤。	5
filter_input_array()	从脚本外部获取多项输入，并进行过滤。	5
filter_list()	返回包含所有得到支持的过滤器的一个数组。	5
filter_var_array()	获取多项变量，并进行过滤。	5
filter_var()	获取一个变量，并进行过滤。	5

41.2 Validating and Sanitizing

有两种过滤器：

- Validating 过滤器：
  - 用于验证用户输入
  - 严格的格式规则（比如 URL 或 E-Mail 验证）
  - 如果成功则返回预期的类型，如果失败则返回 FALSE
- Sanitizing 过滤器：
  - 用于允许或禁止字符串中指定的字符
  - 无数据格式规则
  - 始终返回字符串

41.3 Options and Flags

选项和标志用于向指定的过滤器添加额外的过滤选项。不同的过滤器有不同的选项和标志。

在下面的例子中，我们用 filter_var() 和 “min_range” 以及 “max_range” 选项验证了一个整数：

```
<?php
$var=300;

$int_options = array(
"options"=>array
 (
 "min_range"=>0,
 "max_range"=>256
 )
);

if(!filter_var($var, FILTER_VALIDATE_INT, $int_options))
 {
 echo("Integer is not valid");
 }
else
 {
 echo("Integer is valid");
 }
?>
```

就像上面的代码一样，选项必须放入一个名为 “options” 的相关数组中。如果使用标志，则不需在数组内。这里，由于整数是”300”，它不在指定的范围内，以上代码的输出将是 “Integer is not valid”。

表 41.2: PHP Filters

ID 名称	描述
FILTER_CALLBACK	调用用户自定义函数来过滤数据。
FILTER_SANITIZE_STRING	去除标签，去除或编码特殊字符。
FILTER_SANITIZE_STRIPPED	“string” 过滤器的别名。
FILTER_SANITIZE_ENCODED	URL-encode 字符串，去除或编码特殊字符。
FILTER_SANITIZE_SPECIAL_CHARS	HTML 转义字符'"<>& 以及 ASCII 值小于 32 的字符。
FILTER_SANITIZE_EMAIL	删除所有字符，除了字母、数字以及 !#$%&'*+-/=?^_`{|}~@.[]

ID 名称	描述
FILTER_SANITIZE_URL	删除所有字符，除了字母、数字以及 $-_.+!*'(),{}|\\^~[]`<>#%";/?:@\&=
FILTER_SANITIZE_NUMBER_INT	删除所有字符，除了数字和 +-
FILTER_SANITIZE_NUMBER_FLOAT	删除所有字符，除了数字、+- 以及.,eE。
FILTER_SANITIZE_MAGIC_QUOTES	应用 addslashes()。
FILTER_UNSAFE_RAW	不进行任何过滤，去除或编码特殊字符。
FILTER_VALIDATE_INT	在指定的范围以整数验证值。
FILTER_VALIDATE_BOOLEAN	如果是 “1”, “true”, “on” 以及 “yes”，则返回 true，如果是 “0”, “false”, “off”,“no” 以及 “”，则返回 false。否则返回 NULL。
FILTER_VALIDATE_FLOAT	以浮点数验证值。
FILTER_VALIDATE_REGEXP	根据 regexp，兼容 Perl 的正则表达式来验证值。
FILTER_VALIDATE_URL	把值作为 URL 来验证。
FILTER_VALIDATE_EMAIL	把值作为 e-mail 来验证。
FILTER_VALIDATE_IP	把值作为 IP 地址来验证。

41.4 Validate Input

验证和过滤用户输入或自定义数据是任何 Web 应用程序的重要组成部分，其中输入过滤是最重要的应用程序安全课题之一。

通过使用过滤器，能够确保应用程序获得正确的输入类型，而且应该始终对外部数据进行过滤。

在下面的验证来自表单的输入的示例中，输入变量 “email” 被传到 PHP 页面，我们需要作的第一件事情是确认是否存在我们正在查找的输入数据，然后我们用 filter_input() 函数过滤输入的数据。

```
<?php
if(!filter_has_var(INPUT_GET, "email"))
 {
 echo("Input type does not exist");
 }
else
 {
 if (!filter_input(INPUT_GET, "email", FILTER_VALIDATE_EMAIL))
  {
```

```
 echo "E-Mail is not valid";
 }
else
 {
 echo "E-Mail is valid";
 }
}
?>
```

上面的例子有一个通过”GET” 方法传送的输入变量 (email)：

1. 检测是否存在 “GET” 类型的”email” 输入变量
2. 如果存在输入变量，检测它是否是有效的邮件地址

41.5 Sanitize Input

在下面的清理从表单传来的 URL 的例子中，输入变量 “url” 被传到 PHP 页面，接下来，我们首先要确认是否存在我们正在查找的输入数据，然后用 filter_input() 函数来净化输入数据。

```
<?php
if(!filter_has_var(INPUT_POST, "url"))
 {
 echo("Input type does not exist");
 }
else
 {
 $url = filter_input(INPUT_POST, "url", FILTER_SANITIZE_URL);
 }
?>
```

上面的例子有一个通过”POST” 方法传送的输入变量 (url)：

1. 检测是否存在”POST” 类型的”url” 输入变量
2. 如果存在此输入变量，对其进行净化（删除非法字符），并将其存储在 $url 变量中

假如输入变量类似这样：“http://www.th 非 e 法 qiong.com/”，则净化后的 $url 变量应该是这样的：`http://www.theqiong.com/`

41.6 Filter Multiple Inputs

表单通常由多个输入字段组成。为了避免对 filter_var 或 filter_input 重复调用，我们可以使用 filter_var_array 或 the filter_input_array 函数。

在下面的示例中，我们使用 filter_input_array() 函数来过滤三个 GET 变量。接收到的 GET 变量是一个名字、一个年龄以及一个邮件地址：

```
<?php
$filters = array
 (
 "name" => array
  (
  "filter"=>FILTER_SANITIZE_STRING
  ),
 "age" => array
  (
  "filter"=>FILTER_VALIDATE_INT,
  "options"=>array
   (
   "min_range"=>1,
   "max_range"=>120
   )
  ),
 "email"=> FILTER_VALIDATE_EMAIL,
 );

$result = filter_input_array(INPUT_GET, $filters);

if (!$result["age"])
 {
 echo("Age must be a number between 1 and 120.<br />");
 }
elseif(!$result["email"])
 {
 echo("E-Mail is not valid.<br />");
 }
else
 {
 echo("User input is valid");
 }
?>
```

上面的例子有三个通过"GET" 方法传送的输入变量 (name, age and email)

1. 设置一个数组，其中包含了输入变量的名称，以及用于指定的输入变量的过滤器
2. 调用 filter_input_array 函数，参数包括 GET 输入变量及刚才设置的数组
3. 检测 $result 变量中的 “age” 和 “email” 变量是否有非法的输入。(如果存在非法输入)

filter_input_array() 函数的第二个参数可以是数组或单一过滤器的 ID。如果该参数是单一过滤器的 ID，那么这个指定的过滤器会过滤输入数组中所有的值。

如果该参数是一个数组，那么此数组必须遵循下面的规则：

- 必须是一个关联数组，其中包含的输入变量是数组的键（比如 “age” 输入变量）
- 此数组的值必须是过滤器的 ID ，或者是规定了过滤器、标志以及选项的数组

41.7 Filter Callback

通过使用 FILTER_CALLBACK 过滤器，可以调用自定义的函数，把它作为一个过滤器来使用。这样，我们就拥有了数据过滤的完全控制权。

可以创建自己的自定义函数，也可以使用已有的 PHP 函数，而且规定准备用到过滤器函数的方法，与规定选项的方法相同。

在下面的例子中，我们使用了一个自定义的函数把所有”_” 转换为空格：

```
<?php
function convertSpace($string)
{
return str_replace("_", " ", $string);
}

$string = "Peter_is_a_great_guy!";

echo filter_var($string, FILTER_CALLBACK, array("options"=>"convertSpace"));
?>
```

上面的例子把所有 “_” 转换成空格：

1. 创建一个把 “_” 替换为空格的函数
2. 调用 filter_var() 函数，它的参数是 FILTER_CALLBACK 过滤器以及包含我们的函数的数组

Chapter 42

PHP Error Handling

在创建脚本和 web 应用程序时，错误处理是一个重要的部分。如果代码缺少错误检测编码，那么程序看上去很不专业，也为安全风险敞开了大门。

在 PHP 中，默认的错误处理很简单。一条消息会被发送到浏览器，这条消息带有文件名、行号以及一条描述错误的消息。

PHP 中不同的错误处理方法包括：

- 简单的 “die()” 语句
- 自定义错误和错误触发器
- 错误报告

42.1 Basic Error Handling

在操作文本文件的脚本中，如果文件不存在，会产生错误。为了避免用户获得错误消息，我们在访问文件之前检测该文件是否存在：

```
<?php
if(!file_exists("welcome.txt"))
 {
 die("File not found");
 }
else
 {
 $file=fopen("welcome.txt","r");
 }
?>
```

die() 采用了一个简单的错误处理机制在错误之后终止了脚本。不过，简单地终止脚本并不总是恰当的方式。

42.2 Custom errors and error triggers

error 和 logging 函数允许开发者对错误进行处理和记录，从而使得创建一个自定义的错误处理器非常简单，其中：

- error 函数允许用户定义错误处理规则，并修改记录错误的方式。
- logging 函数允许用户对应用程序进行日志记录，并把日志消息发送到电子邮件、系统日志或其他的机器。

下面我们很简单地创建了一个专用函数，可以在 PHP 中发生错误时调用该函数。

该函数必须有能力处理至少两个参数 (error level 和 error message)，但是可以接受最多五个参数（可选的：file, line-number 以及 error context)：

```
error_function(error_level,error_message,
error_file,error_line,error_context)
```

表 42.1: PHP 自定义错误处理器

参数	描述
error_level	必需。为用户定义的错误规定错误报告级别。必须是一个值数。
error_message	必需。为用户定义的错误规定错误消息。
error_file	可选。规定错误在其中发生的文件名。
error_line	可选。规定错误发生的行号。
error_context	可选。规定一个数组，包含了当错误发生时在用的每个变量以及它们的值。

42.3 Error Reporting

下面这些错误报告级别是错误处理程序旨在处理的错误的不同的类型：

表 42.2: PHP 错误报告级别

值	常量	描述
2	E_WARNING	非致命的 run-time 错误。不暂停脚本执行。
8	E_NOTICE	Run-time 通知。脚本发现可能有错误发生，但也可能在脚本正常运行时发生。
256	E_USER_ERROR	致命的用户生成的错误。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_ERROR。

值	常量	描述
512	E_USER_WARNING	非致命的用户生成的警告。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_WARNING。
1024	E_USER_NOTICE	用户生成的通知。这类似于程序员使用 PHP 函数 trigger_error() 设置的 E_NOTICE。
4096	E_RECOVERABLE_ERROR	可捕获的致命错误。类似 E_ERROR，但可被用户定义的处理程序捕获。(参见 set_error_handler())
8191	E_ALL[1]	所有错误和警告，除级别 E_STRICT 以外。

下面的代码创建一个处理错误的函数：

```
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }
```

当这个错误处理函数被触发时，它会取得错误级别和错误消息。然后它会输出错误级别和消息，并终止脚本。

创建了一个错误处理函数之后，下一步我们需要确定的是在何时触发该函数。

42.4 Error Handler

PHP 的默认错误处理程序是内建的错误处理程序，而且 PHP 允许将自定义错误处理函数改造为脚本运行期间的默认错误处理程序。

还可以修改错误处理程序，使其仅应用到某些错误，这样脚本就可以不同的方式来处理不同的错误。不过，下面将针对所有错误来使用我们的自定义错误处理程序：

```
set_error_handler("customError");
```

由于我们希望自定义函数来处理所有错误，于是 set_error_handler() 仅使用一个参数，可以添加第二个参数来规定错误级别。下面通过尝试输出不存在的变量，来测试这个错误处理程序：

```
<?php
//error handler function
function customError($errno, $errstr)
```

[1]在 PHP 6.0，E_STRICT 是 E_ALL 的一部分。

```
{
echo "<b>Error:</b> [$errno] $errstr";
}

//set error handler
set_error_handler("customError");

//trigger error
echo($test);
?>
```

以上代码的输出应该类似这样：`Error: [8] Undefined variable: test`

42.5 Error Trigger

在脚本中用户输入数据的位置，当用户的输入无效时触发错误的很有用的。在 PHP 中，这个任务由 trigger_error() 完成。

在下面的示例中，如果 “test” 变量大于 “1”，就会发生错误：

```
<?php
$test=2;
if ($test>1)
{
trigger_error("Value must be 1 or below");
}
?>
```

可以在脚本中任何位置触发错误，通过添加的第二个参数，能够规定所触发的错误级别。

可能的错误类型包括：

- E_USER_ERROR - 致命的用户生成的 run-time 错误。错误无法恢复。脚本执行被中断。
- E_USER_WARNING - 非致命的用户生成的 run-time 警告。脚本执行不被中断。
- E_USER_NOTICE - 默认。用户生成的 run-time 通知。脚本发现了可能的错误，也有可能在脚本运行正常时发生。

在接下来的例子中，如果 “test” 变量大于 “1”，则发生 E_USER_WARNING 错误。如果发生了 E_USER_WARNING，我们将使用我们的自定义错误处理程序并结束脚本：

```
<?php
//error handler function
function customError($errno, $errstr)
{
```

```
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Ending Script";
 die();
 }

//set error handler
set_error_handler("customError",E_USER_WARNING);

//trigger error
$test=2;
if ($test>1)
 {
 trigger_error("Value must be 1 or below",E_USER_WARNING);
 }
?>
```

42.6 Error Logging

默认地，根据在 php.ini 中的 error_log 配置，PHP 向服务器的错误记录系统或文件发送错误记录。通过使用 error_log() 函数，可以向指定的文件或远程目的地发送错误记录，其中通过电子邮件向自己发送错误消息，是一种获得指定错误的通知的好办法。

在下面的例子中，如果特定的错误发生，我们将发送带有错误消息的电子邮件，并结束脚本：

```
<?php
//error handler function
function customError($errno, $errstr)
 {
 echo "<b>Error:</b> [$errno] $errstr<br />";
 echo "Webmaster has been notified";
 error_log("Error: [$errno] $errstr",1,
 "someone@example.com","From: webmaster@example.com");
}

//set error handler
set_error_handler("customError",E_USER_WARNING);

//trigger error
$test=2;
if ($test>1)
```

```
{
trigger_error("Value must be 1 or below",E_USER_WARNING);
}
?>
```

以上代码的输出应该类似这样：

```
Error: [512] Value must be 1 or below
Webmaster has been notified
```

接收自以上代码的邮件类似这样：

```
Error: [512] Value must be 1 or below
```

但是这个方法不适合所有的错误，常规错误还是应当通过使用默认的 PHP 记录系统在服务器上进行记录。

42.7 PHP Error and Logging Functions

error 和 logging 函数是 PHP 核心的组成部分，无需安装即可使用这些函数。

表 42.3: PHP Error 和 Logging 函数

函数	描述	PHP
debug_backtrace()	生成 backtrace。	4
debug_print_backtrace()	输出 backtrace。	5
error_get_last()	获得最后发生的错误。	5
error_log()	向服务器错误记录、文件或远程目标发送一个错误。	4
error_reporting()	规定报告哪个错误。	4
restore_error_handler()	恢复之前的错误处理程序。	4
restore_exception_handler()	恢复之前的异常处理程序。	5
set_error_handler()	设置用户自定义的错误处理函数。	4
set_exception_handler()	设置用户自定义的异常处理函数。	5
trigger_error()	创建用户自定义的错误消息。	4
user_error()	trigger_error() 的别名。	4

42.8 PHP Error and Logging Constants

表 42.4: PHP Error 和 Logging 常量

值	常量	描述	PHP
1	E_ERROR	致命的运行时错误。错误无法恢复。脚本的执行被中断。	
2	E_WARNING	非致命的运行时错误。脚本的执行不会中断。	
4	E_PARSE	编译时语法解析错误。解析错误只应该由解析器生成。	
8	E_NOTICE	运行时提示。可能是错误，也可能在正常运行脚本时发生。	
16	E_CORE_ERROR	由 PHP 内部生成的错误。	4
32	E_CORE_WARNING	由 PHP 内部生成的警告。	4
64	E_COMPILE_ERROR	由 Zend 脚本引擎内部生成的错误。	4
128	E_COMPILE_WARNING	由 Zend 脚本引擎内部生成的警告。	4
256	E_USER_ERROR	由于调用 trigger_error() 函数生成的运行时错误。	4
512	E_USER_WARNING	由于调用 trigger_error() 函数生成的运行时警告。	4
1024	E_USER_NOTICE	由于调用 trigger_error() 函数生成的运行时提示。	4
2048	E_STRICT	运行时提示。对增强代码的互用性和兼容性有益。	5
4096	E_RECOVERABLE_ERROR[2]	可捕获的致命错误。	5
8191	E_ALL	所有的错误和警告，除了 E_STRICT。	5

[2]参阅 set_error_handler()

Part VI

References

Chapter 43

Overview

在 PHP 中引用意味着用不同的名字访问同一个变量内容。

PHP 的引用和 C 的指针不同，这里的引用是符号表别名，而且 PHP 中的变量名和变量内容是不一样的，因此同样的内容可以有不同的名字。

最接近 PHP 的引用的比喻是 Unix 的文件名和文件本身——变量名是目录条目，而变量内容则是文件本身，因此引用可以被看作是 Unix 文件系统中的 hardlink。

PHP 的引用允许用两个变量来指向同一个内容。例如，在下面的示例中的 $a 和 $b 指向了同一个变量。

```
<?php
$a =& $b;
?>
```

$a 和 $b 在这里是完全相同的，这并不是 $a 指向了 $b 或者相反，而是 $a 和 $b 指向了同一个地方。

- 如果具有引用的数组被拷贝，其值不会解除引用。对于数组传值给函数也是如此。
- 如果对一个未定义的变量进行引用赋值、引用参数传递或引用返回，则会自动创建该变量。

Example 31　对未定义的变量使用引用

```
<?php
function foo(&$var) { }

foo($a); // $a is "created" and assigned to null

$b = array();
foo($b['b']);
var_dump(array_key_exists('b', $b)); // bool(true)
```

```
$c = new StdClass;
foo($c->d);
var_dump(property_exists($c, 'd')); // bool(true)
?>
```

同样的语法可以用在函数中来返回引用，或者用在 new 运算符中。

```
<?php
$bar =& new fooclass();
$foo =& find_var($bar);
?>
```

PHP 的 new 操作自动返回引用，现在再使用 =& 已经过时了并且会产生 E_STRICT 级别的消息。

不用 & 运算符导致对象生成了一个拷贝。如果在类中用 $this，它将作用于该类当前的实例。没有用 & 的赋值将拷贝这个实例（例如对象）并且 $this 将作用于这个拷贝上，这并不总是想要的结果。由于性能和内存消耗的问题，通常只想工作在一个实例上面。

尽管可以用 @ 运算符来抑制构造函数中的任何错误信息（例如用 @new），但用 &new 语句时这不起效果。这是 Zend 引擎的一个限制并且会导致一个解析错误。

如果在一个函数内部给一个声明为 global 的变量赋于一个引用，该引用只在函数内部可见。可以通过使用 $GLOBALS 数组避免这一点。

Example 32 在函数内引用全局变量

```
<?php
$var1 = "Example variable";
$var2 = "";

function global_references($use_globals)
{
    global $var1, $var2;
    if (!$use_globals) {
        $var2 =& $var1; // visible only inside the function
    } else {
        $GLOBALS["var2"] =& $var1; // visible also in global context
    }
}

global_references(false);
echo "var2 is set to '$var2'\n"; // var2 is set to ''
global_references(true);
```

```
echo "var2 is set to '$var2'\n"; // var2 is set to 'Example variable'
?>
```

把 `global $var;` 当成是 `$var =& $GLOBALS['var'];` 的简写，从而将其它引用赋给 $var 只改变了本地变量的引用。

如果在 foreach 语句中给一个具有引用的变量赋值，被引用的对象也被改变。

Example 33　引用与 foreach 语句

```
<?php
$ref = 0;
$row =& $ref;
foreach (array(1, 2, 3) as $row) {
    // do something
}
echo $ref; // 3 - last element of the iterated array
?>
```

引用做的第二件事是用引用传递变量。这是通过在函数内建立一个本地变量并且该变量在呼叫范围内引用了同一个内容来实现的。

```
<?php
function foo(&$var)
{
    $var++;
}

$a=5;
foo($a);
?>
```

将使 $a 变成 6。这是因为在 foo 函数中变量 $var 指向了和 $a 指向的同一个内容。

引用做的第三件事是引用返回，而且引用不是指针，因此意味着下面的结构不会产生预期的效果：

```
<?php
function foo(&$var)
{
    $var =& $GLOBALS["baz"];
}
foo($bar);
?>
```

这将使 foo 函数中的 $var 变量在函数调用时和 $bar 绑定在一起，但接着又被重新绑定到了 $GLOBALS[”baz”] 上面。不可能通过引用机制将 $bar 在函数调用范围内绑定到别的变量上面，因为在函数 foo 中并没有变量 $bar（它被表示为 $var，但是 $var 只有变量内容而没有调用符号表中的名字到值的绑定)。

引用返回可以被用来引用被函数选择的变量。

Chapter 44

Passing by Reference

可以将一个变量通过引用传递给函数，这样该函数就可以修改其参数的值。

```
<?php
function foo(&$var)
{
    $var++;
}

$a=5;
foo($a);
// $a is 6 here
?>
```

注意，在函数调用时没有引用符号——只有函数定义中有。

仅仅是函数定义就足够使参数通过引用来正确传递了，因此如果把 & 用在 foo(&$a); 中会得到一条警告说“Call-time pass-by-reference”已经过时了。

以下内容可以通过引用传递：

- 变量，例如 foo($a)
- new 语句，例如 foo(new foobar())
- 从函数中返回的引用，例如：

```
<?php
function &bar()
{
    $a = 5;
    return $a;
}
foo(bar());
```

```
?>
```

任何其它表达式都不能通过引用传递，结果未定义。例如，下面引用传递的例子是无效的：

```
<?php
function bar() // Note the missing &
{
   $a = 5;
   return $a;
}
foo(bar()); // 自 PHP 5.0.5 起导致致命错误
foo($a = 5) // 表达式，不是变量
foo(5) // 导致致命错误
?>
```

Chapter 45

Returning References

引用返回用在当想用函数找到引用应该被绑定在哪一个变量上面时。不要用返回引用来增加性能，引擎足够聪明来自己进行优化，仅在有合理的技术原因时才返回引用。

```
<?php
class foo {
    public $value = 42;

    public function &getValue() {
        return $this->value;
    }
}

$obj = new foo;
$myValue = &$obj->getValue(); // $myValue is a reference to $obj->value, which is 42.
$obj->value = 2;
echo $myValue;            // prints the new value of $obj->value, i.e. 2.
?>
```

本例中 getValue 函数所返回的对象的属性将被赋值，而不是拷贝，就和没有用引用语法一样。

和参数传递不同，这里必须在两个地方都用 & 符号——指出返回的是一个引用，而不是通常的一个拷贝，同样也指出 $myValue 是作为引用的绑定，而不是通常的赋值。

如果试图这样从函数返回引用：`return ($this->value);` 将不会起作用，因为在试图返回一个表达式的结果而不是一个引用的变量，只能从函数返回引用变量——没别的方法。

如果代码试图返回一个动态表达式或 new 运算符的结果会产生出一条 E_NOTICE 错误。

Chapter 46

Unsetting References

unset 一个引用只是断开了变量名和变量内容之间的绑定，不过这并不意味着变量内容被销毁了。

```
<?php
$a = 1;
$b =& $a;
unset($a);
?>
```

不会 unset $b，只是 $a，可以类比于 Unix 的 unlink 调用。

```
<?php
/* Imagine this is memory map
 ------------------------------
|pointer | value | variable        |
 ----------------------------------
|   1    |  NULL |        ---         |
|   2    |  NULL |        ---         |
|   3    |  NULL |        ---         |
|   4    |  NULL |        ---         |
|   5    |  NULL |        ---         |
-----------------------------------
Create some variables */
$a=10;
$b=20;
$c=array ('one'=>array (1, 2, 3));
/* Look at memory
 ------------------------------
|pointer | value |    variable's     |
```

```
 ----------------------------------
|   1    | 10    |      $a            |
|   2    | 20    |      $b            |
|   3    | 1     |     $c['one'][0] |
|   4    | 2     |     $c['one'][1] |
|   5    | 3     |     $c['one'][2] |
-----------------------------------
do */
$a=&$c['one'][2];
/* Look at memory
 --------------------------------
|pointer | value |    variable's    |
 ----------------------------------
|   1    | NULL |      ---           | //value of $a is destroyed and pointer is free
|   2    | 20    |      $b            |
|   3    | 1     |     $c['one'][0] |
|   4    | 2     |     $c['one'][1] |
|   5    | 3     | $c['one'][2] ,$a | // $a is now here
-----------------------------------
do */
$b=&$a; // or $b=&$c['one'][2]; result is same as both "$c['one'][2]" and "$a" is at
    same pointer.
/* Look at memory
 --------------------------------
|pointer | value |    variable's       |
 -------------------------------------
|   1    | NULL |      ---              |
|   2    | NULL |      ---              | //value of $b is destroyed and pointer is
    free
|   3    | 1     |     $c['one'][0]    |
|   4    | 2     |     $c['one'][1]    |
|   5    | 3     |$c['one'][2] ,$a , $b | // $b is now here
-------------------------------------
next do */
unset($c['one'][2]);
/* Look at memory
 --------------------------------
|pointer | value |    variable's       |
 -------------------------------------
|   1    | NULL |      ---              |
|   2    | NULL |      ---              |
```

```
|  3    | 1      |     $c['one'][0]    |
|  4    | 2      |     $c['one'][1]    |
|  5    | 3      |     $a , $b          | // $c['one'][2] is destroyed not in memory,
    not in array
---------------------------------------
next do  */
$c['one'][2]=500; //now it is in array
/* Look at memory
 ---------------------------------
|pointer | value |   variable's        |
 ---------------------------------------
|  1    | 500   |     $c['one'][2]    | //created it lands on any(next) free pointer
    in memory
|  2    | NULL |      ---              |
|  3    | 1      |     $c['one'][0]    |
|  4    | 2      |     $c['one'][1]    |
|  5    | 3      |     $a , $b          | //this pointer is in use
---------------------------------------
lets tray to return $c['one'][2] at old pointer an remove reference $a,$b. */
$c['one'][2]=&$a;
unset($a);
unset($b);
/* look at memory
 ---------------------------------
|pointer | value |   variable's        |
 ---------------------------------------
|  1    | NULL |      ---              |
|  2    | NULL |      ---              |
|  3    | 1      |     $c['one'][0]    |
|  4    | 2      |     $c['one'][1]    |
|  5    | 3      |     $c['one'][2]    | //$c['one'][2] is returned, $a,$b is destroyed
---------------------------------------
?>
```

Chapter 47

Spotting References

许多 PHP 的语法结构（例如引用传递和返回）都是通过引用机制实现的，所以上述有关引用绑定的一切也都适用于这些结构。

- 引用传递
- 引用返回
- global 引用
- $this

47.1 global References

使用 `global $var;` 声明一个变量时实际上建立了一个到全局变量的引用，和注册到全局变量 `$GLOBALS` 相同。

```
<?php
$var =& $GLOBALS["var"];
?>
```

`unset $var` 不会 unset 全局变量。

47.2 $this

在一个对象的方法中，$this 永远是调用它的对象的引用。

Part VII

SPL

Chapter 48

Overview

SPL（Standard PHP Library）扩展是用于解决典型问题 (standard problems) 的一组接口与类的集合。

48.1 Requirement

SPL 扩展不需要其他扩展，而且作为 PHP 内核组件的一部分，SPL 是一直有效的。

48.2 Runtime Configuration

SPL 扩展没有在 php.ini 中定义配置指令。

48.3 Resource Type

SPL 扩展没有定义资源类型。

48.4 Predefined Constants

SPL 扩展预定义的常量仅在此扩展编译入 PHP 或在运行时动态载入时可用。
在 SPL 使用类常量之前使用的在 RIT_LEAVES_ONLY 中定义的全局常量。

Chapter 49

Data Structure

SPL 提供了一组标准数据结构。

49.1 SplDoublyLinkedList

双向链表

49.2 SplStack

49.3 SplQueue

49.4 SplHeap

堆

49.5 SplMaxHeap

49.6 SplMinHeap

49.7 SplPriorityQueue

49.8 SplFixedArray

阵列

49.9 SplObjectStorage

映射

Chapter 50

Iterator

SPL 提供一系列迭代器以遍历不同的对象。

50.1 AppendIterator

50.2 ArrayIterator

50.3 CachingIterator

50.4 CallbackFilterIterator

50.5 DirectoryIterator

50.6 EmptyIterator

50.7 FilesystemIterator

50.8 FilterIterator

50.9 GlobIterator

50.10 InfiniteIterator

50.11 LimitIterator

50.12 MultipleIterator

50.13 NoRewindIterator

50.14 ParentIterator

50.15 RecursiveArrayIterator

50.16 RecursiveCachingIterator

50.17 RecursiveCallbackIterator

50.18 RecursiveDirectoryIterator

50.19 RecursiveFilterIterator

50.20 RecursiveIteratorIterator

50.21 RecursiveRegexIterator

50.22 RecursiveTreeIterator

Chapter 51

Interfaces

SPL 提供一系列接口。

51.1 Countable

51.2 OuterIterator

51.3 RecursiveIterator

51.4 SeekableIterator

51.5 SplObserver

51.6 SplSubject

Chapter 52

Exception

SPL 提供一系列标准异常。

Chapter 53

Classes

Chapter 54

Functions

Part VIII

Database

Chapter 55

Overview

55.1 Native driver

MySQL 不仅是事实上的标准数据库（比如 Friendster, Yahoo, Google），还可以进行缩减来支持嵌入的数据库应用程序。

绝大多数的 PHP 应用都需要使用数据库来进行数据持久化，PHP 支持使用 mysqli、pgsql 和 mssql 等原生驱动来连接数据库进行交互。

原生驱动是在只使用一个数据库的情况下的不错的方式，而且 MySQLi 模块还提供了更加有效的方法和实用工具来处理数据库操作，这些方法大都是以面向对象的方式实现。

除了 mysqli 之外，PHP 还内置了一个数据库连接抽象类库——PDO 来提供了一个通用的接口来与不同的数据库进行交互。例如，可以使用相同的简单代码来连接 MySQL 或是 SQLite：

```
<?php
// PDO + MySQL
$pdo = new PDO('mysql:host=example.com;dbname=database', 'user', 'password');
$statement = $pdo->query("SELECT some_field FROM some_table");
$row = $statement->fetch(PDO::FETCH_ASSOC);
echo htmlentities($row['some_field']);

// PDO + SQLite
$pdo = new PDO('sqlite:/path/db/foo.sqlite');
$statement = $pdo->query("SELECT some_field FROM some_table");
$row = $statement->fetch(PDO::FETCH_ASSOC);
echo htmlentities($row['some_field']);
```

PDO 并不会对 SQL 请求进行转换或者模拟实现并不存在的功能特性，它只是单纯地使用相同的 API 连接不同种类的数据库。

更重要的是，PDO 使用户能够安全的插入外部输入（例如 ID）到 SQL 请求中而不必担心 SQL 注入的问题，可以通过使用 PDO 语句和限定参数来实现。

假设一个 PHP 脚本接收一个数字 ID 作为一个请求参数。这个 ID 应该被用来从数据库中取出一条用户记录。

首先，下面是一个错误的做法：

```
<?php
$pdo = new PDO('sqlite:/path/db/users.db');
$pdo->query("SELECT name FROM users WHERE id = " . $_GET['id']); // <-- NO!
```

插入一个原始的请求参数到 SQL 请求中可以让被黑客轻松地利用 SQL 注入方式进行攻击。例如，如果黑客将一个构造的 id 参数通过类似 `http://domain.com/?id=1%3BDELETE+FROM+users` 的 URL 传入，就会使 `$_GET['id']` 变量的值被设置为 `1;DELETE FROM users` 然后被执行从而删除所有的 user 记录，因此，必须使用 PDO 限制参数来过滤 ID 输入。

```
<?php
$pdo = new PDO('sqlite:/path/db/users.db');
$stmt = $pdo->prepare('SELECT name FROM users WHERE id = :id');
$id = filter_input(INPUT_GET, 'id', FILTER_SANITIZE_NUMBER_INT);
// <-- filter your data first (see [Data Filtering](#data_filtering)), especially
    important for INSERT, UPDATE, etc.
$stmt->bindParam(':id', $id, PDO::PARAM_INT); // <-- Automatically sanitized for SQL by
    PDO
$stmt->execute();
```

上述代码在一条 PDO 语句中使用了一个限制参数，并对外部 ID 输入在发送给数据库之前进行转义来防止潜在的 SQL 注入攻击。

对于写入操作（例如 INSERT 或者 UPDATE）进行数据过滤并对其他内容进行清理（去除 HTML 标签和 Javascript 等）是尤其重要的，因为 PDO 只会为 SQL 进行清理，并不会为 PHP 代码进行任何处理。

如果数据库连接没有被隐式地关闭，那么数据库连接有可能会耗尽全部资源，有可能会造成可用资源枯竭的情况。

用户使用 PDO 可以销毁（destroy）对象（也就是将值设为 NULL）来隐式地关闭这些连接，以确保所有剩余的引用对象的连接都被删除。如果用户没有亲自清理数据库连接，PHP 在脚本结束的时候自动清理——除非用户使用的是持久链接。

55.2 Abstract layer

当原始的 PHP 应用开发中，通常会将数据库的交互与表示层逻辑混在一起，例如：

```
<ul>
```

```
<?php
foreach ($db->query('SELECT * FROM table') as $row) {
   echo "<li>".$row['field1']." - ".$row['field2']."</li>";
}
?>
</ul>
```

很多方面现在来看都是错误的做法，主要是由于代码不易阅读又难以测试和调试，而且如果不加以限制，实际上会输出非常多的字段。

其实还有许多不同的解决方案来完成这项工作—取决于用户倾向于面向对象编程还是函数式编程—但是必须有一些分离的元素。例如，将两个元素放入了两个不同的文件来得到一些干净的分离，但是仍然是糟糕的做法。

```
<?php
function getAllFoos($db) {
   return $db->query('SELECT * FROM table');
}
foreach (getAllFoos($db) as $row) {
   echo "<li>".$row['field1']." - ".$row['field2']."</li>"
}
?>
```

接下来通过进一步改进来创建一个类来放置上面的函数，于是得到一个「Model」，同时创建一个简单的.php 文件来存放表示逻辑，这样就得到了一个「View」，整个过程已经很接近 MVC 架构。

- foo.php

```
<?php
$db = new PDO('mysql:host=localhost;dbname=testdb;charset=utf8', 'username', '
    password');

// Make your model available
include 'models/FooModel.php';

// Create an instance
$fooModel = new FooModel($db);
// Get the list of Foos
$fooList = $fooModel->getAllFoos();

// Show the view
include 'views/foo-list.php';
```

- models/FooModel.php

```
<?php
class FooModel
{
    protected $db;

    public function __construct(PDO $db)
    {
        $this->db = $db;
    }

    public function getAllFoos() {
        return $this->db->query('SELECT * FROM table');
    }
}
```

- views/foo-list.php

```
<?php foreach ($fooList as $row): ?>
    <?= $row['field1'] ?> - <?= $row['field1'] ?>
<?php endforeach ?>
```

向大多数现代框架的做法学习是很有必要的，尽管多了一些手动的工作，这样就不需要每一次都完全这么做，但是将太多的表示逻辑层代码和数据库交互掺杂在一些将会对程序进行单元测试时带来真正的麻烦，于是 PHPBridge 提供了一类资源叫做数据类，其中包含了非常相似的适合开发使用的数据库操作逻辑。

许多框架都提供了自己的数据库抽象层，其中一些是设计在 PDO 的上层的。

数据库抽象层通常将请求在 PHP 方法中包装起来，通过模拟的方式来使数据库拥有一些之前不支持的功能，因此这种抽象是真正的数据库抽象，而不单单只是 PDO 提供的数据库连接抽象。

数据库抽象的确会增加一定程度的性能开销，但是如果应用程序需要同时使用 MySQL，PostgreSQL 和 SQLite 时，这些额外的性能开销对于代码整洁度的提高来说还是很值得的。

数据库抽象层通常使用的是 PSR-0 或 PSR-4 命名空间标准，所以可以把它们安装在任何应用程序中。

- Aura SQL(https://github.com/auraphp/Aura.Sql)
- Doctrine2 DBAL(https://github.com/doctrine/dbal)
- Propel(https://github.com/propelorm/Propel2)
- ZF2 Db(https://github.com/zendframework/zend-db)

55.3 mysql_connect()

在能够访问并处理数据库中的数据之前，必须创建到达数据库的连接。在 PHP 中，这个任务通过 mysql_connect() 函数完成。

```
mysql_connect(servername,username,password);
```

- servername
  可选，规定要连接的服务器，默认是 “localhost:3306”。
- username
  可选，规定登录所使用的用户名，默认值是拥有服务器进程的用户的名称。
- password
  可选，规定登录所用的密码，默认是 “”。

下面的脚本中使用变量 $con 存放数据库连接。如果连接失败，将执行 “die” 部分：

Example 34 mysql_connect()

```
<?php
  $con = mysql_connect("localhost","admin","testtest");
  if(!$con){
    die('Could not connect: ' . mysql_error() );
  }
  //some code
?>
```

55.4 mysql_pconnect()

PHP 的 MySQL 持久化连接，美好的目标，却拥有糟糕的口碑。

作为 Apache 模块运行的 PHP 来说，要实现 MySQL 持久化连接，首先得取决于 Apache 是否支持 Keep-Alive。

从客户在浏览器输入一个有效 url 地址开始，浏览器就会利用 socket 向 url 对应的 web 服务器发送一条 TCP 请求，这个请求成功一次就得需要来回握三次手才能确定，成功以后，浏览器利用 socket TCP 连接资源向 web 服务器请求 http 协议，发送以后就等着 web 服务器把 http 返回头和 body 发送回来，发回来后浏览器关闭 socket 连接，然后做 http 返回头和 body 的解析工作，最后呈现在浏览器上的就是页面。

TCP 连接需要三次握手，也就是来回请求三次方能确定一个 TCP 请求是否成功，TCP 关闭来回需要 4 次请求才能完成。

如果 Web 服务器没有开启 Keep-Alive，那么每次 http 请求都需要 3 次握手和 4 次通信，很多时间和资源消耗在在 socket 连接关闭上，从 HTTP 1.0 开始引入的 Keep-Alive 允许

一次 socket TCP 连接来发送多次 http 请求，不过 http/1.0 里需要客户端自己在请求头加入 Connection:Keep-alive 参数。

Apache 从 1.2 版本开始支持 Keep-Alive，可以在 httpd.conf 中设置 KeepAlive 配置项为 On，MaxKeepAliveRequests 为 0。

MaxKeepAliveRequests 指定一个持久 TCP 最多允许的请求数，如果过小就很容易在 TCP 未过期的情况下到达最大连接，那么下次连接就已经是新的 TCP 连接，这里设置为 0 表示不限制，并且在 mysql_pconnect() 中设置 KeepAliveTimeout 为 15 秒。

Example 35 获取当前 PHP 执行者 (Apache) 的进程号

```
<?php
echo "Apache进程号:". getmypid();
```

如果在 15 秒内连续刷新页面就发现进程号不变，但是超过 15 秒则进程号会变化。

在 Apache 里设置 KeepAliveTimeout 为 15 秒，那么在 Web 服务器默认打开 KeepAlive 的情况下，客户端第一次 http 成功请求后，Apache 不会立刻断开 socket，而是一直监听来自这一客户端的请求，并且根据 KeepAliveTimeout 选项配置的时间决定是否端口连接。

Apache 在 socket 连接超时后就会关闭 socket，下次同一客户端如果再次请求，Apache 就会新开一个进程来响应，所以在 15 秒内刷新页面时看到的进程号是一致的，表明是浏览器请求被转发给了同一个 Apache 进程。

http 返回头里带上 Connection:keep-alive 和 Keep-alive:15 就会让客户端浏览器知道本次 socket 连接并未关闭，而且可以在 15 秒内继续使用当前这个 socket 连接来发送 http 请求。

mysql_pconnect 根据当前 Apache 的进程号来生成 hash key，并查找 hash 表内有无对应的连接资源，如果没有则压入 hash 表，否则就直接使用。

Example 36 ext/mysql/PHP_mysql.c 中的 PHP_mysql_do_connect() 函数

```
#1.生成hash key
user=php_get_current_user();//获取当前PHP执行者(Apache)的进程唯一标识号
//hashed_details就是hash key
hashed_details_length = spprintf(&hashed_details, 0, "MySQL__%s_", user);
  #2.如果未找到已有资源，就推入hash表，名字叫persistent_list，如果找到就直接使
      用
/* try to find if we already have this link in our persistent list */
if (zend_hash_find(&EG(persistent_list), hashed_details,
    hashed_details_length+1, (void **) &le)==FAILURE) {
 /* we don't */
 ...
 ...
 /* hash it up(推入hash表) */
```

```
Z_TYPE(new_le) = le_plink;
new_le.ptr = mysql;
if (zend_hash_update(&EG(persistent_list), hashed_details,
    hashed_details_length+1, (void *) &new_le, sizeof(zend_rsrc_list_entry
    ), NULL)==FAILURE) {
  ...
     ...
  }
    }
else
{/* The link is in our list of persistent connections(连接已在hash表里)*/
        ...
        ...
        mysql = (PHP_mysql_conn *) le->ptr;//直接使用对应的sql连接资源
        ...
        ...
}
```

影响 mysql_pconnect 最重要的两个参数是 wait_timeout 和 interactive_timeout，如果在页面上打印 MySQL 线程号和 Apache 进程号，就会发现在连接超时后二者都变成新的了，原因就是现在已经是新的 Apache 进程在处理请求，进程 id 是新的，hash key 随之变化，这种情况下 PHP 只能重新连接 MySQL，并把连接资源压入 persistent list。

```
<?php
$conn = mysql_pconnect("localhost","root","123456") or die("Can not connect to MySQL");
echo "MySQL线程号:". MySQL_thread_id($conn). "<br />";
echo "Apache进程号". getmypid();
```

MySQL_thread_id 是 MySQL 中正在执行的线程的号码，每个线程都有固定的 wait_timeout 和 interactive_timeout，默认都是 8 小时，因此很容易导致 processlist 堆积并报出 Too many connections 错误，即使 max_connections 设置的再大也无济于事。

```
mysql> show processlist;
+-----+------+-----------+------+--------+-----+------+-----------------+
| Id  | User | Host      | db   | Command| Time| State| Info            |
+-----+------+-----------+------+--------+-----+------+-----------------+
| 348 | root | localhost | NULL | Query  |    0| NULL | show processlist|
| 349 | root | localhost | NULL | Sleep  |    2|      | NULL            |
+-----+------+-----------+------+--------+-----+------+-----------------+
```

在 my.cnf 中调整 wait_timeout 和 interactive_timeout 的配置值，其中 interactive_timeout 只对 MySQL 的交互式 Shell 有效。

如果设置了 mysql_pconnect() 的第四个参数 MYSQL_CLIENT_INTERACTIVE，那么 MySQL 就会将其视作 interactive connection，这样即使 PHP 页面没有活动，MySQL 也会根据超时时间及时断开超时线程持有的连接。

```
<?php
$conn = mysql_pconnect('localhost','root','123456',MySQL_CLIENT_INTERACTIVE);
echo "MySQL线程号:".MySQL_thread_id($conn)."<br />";
echo "Apache进程号:".getmypid();
```

在使用 PHP 的 mysql_pconnect() 函数时，首先必须保证 Apache 是支持 keep-alive 的，其次 KeepAliveTimeOut 应该设置多久要根据自身站点的访问情况做调整，时间太短则 keep alive 没有意义，时间太长就很可能为一个闲客户端连接牺牲过多的服务器资源，毕竟保持 socket 监听进程是要消耗 CPU 和内存的。

Apache 的 KeepAliveTimeOut 配置需要和 MySQL 的 timeout 配置进行平衡，假设 mysql_pconnect 未带上第 4 个参数，如果 Apache 的 KeepAliveTimeOut 设置的秒数比 wait_timeout 小，那么真正对 mysql_pconnect 起作用的是 Apache 而不是 MySQL 的配置。

如果 MySQL 的 wait_timeout 偏大，在并发量大的情况下很可能开启过多的 connection，而且 MySQL 无法及时回收就会导致 Too many connections 错误。

如果 KeepAliveTimeOut 设置的太大同样会导致无效连接过多，因此 Apache 的 KeepAliveTimeOu 设置不能太大，只要比 MySQL 的 wait_timeout 稍大或者相等就是比较好的方案，这样可以保证 keep alive 过期后，废弃的 MySQL 连接可以及时被回收。

55.5 mysql_close()

脚本执行结束，就会关闭数据库连接。若要提前关闭连接，使用 mysql_close() 函数。

Example 37 mysql_close()

```
<?php
  $con = mysql_connect("localhost","admin","testtest");
  if(!$con){
    die('Could not connect: ' . mysql_error() );
  }
  //some code

  mysql_close($con);
?>
```

55.6 mysql_query()

为了让 PHP 执行 MySQL 数据库操作语句，必须使用 mysql_query() 函数，此函数用于向 MySQL 连接发送查询或命令。

CREATE DATABASE 语句用于在 MySQL 中创建数据库。

```
CREATE DATABASE database_name
```

SQL 语句对大小写不敏感，创建 MySQL 数据库时必须使用 mysql_query() 函数。在下面的示例创建了一个名为 “my_db” 的数据库：

Example 38 创建数据库

```
<?php
  $con = mysql_connect("localhost","admin","testtest");
  if(!$con){
    die('Could not connect ' . mysql_error() );
  }
  //Create database
  if(mysql_query("CREATE DATABASE my_db",$con) ){
    echo "Database created.";
  }else{
    echo "Error creating database: " . mysql_error();
  }

  mysql_close($con);
?>
```

CREATE TABLE 用于在 MySQL 中创建数据库表。

```
CREATE TABLE table_name
(
  column_name1 data_type,
  column_name2 data_type,
  column_name3 data_type,
  ... ...
)
```

为了创建表，必须向 mysql_query() 函数添加 CREATE TABLE 语句。下面的示例创建了一个名为 “Persons” 的表，而且该表中有三列，列名分别是 “FirstName”, “LastName” 以及 “Age”：

Example 39 创建表

```php
<?php
  $con = mysql_connect("localhost","admin","testtest");
  if(!$con){
    die('Could not connect: ' . mysql_error() );
  }
  //Create database
  if(mysql_query("CREATE DATABASE my_db"),$con){
    echo "Database created.";
  }else{
    echo "Error creating database: " . mysql_error();
  }
  //Create table in my_db database
  mysql_select_db("my_db",$con);
  $sql = "CREATE TABLE Persons
  (
    FirstName varchar(15),
    LastName varchar(15),
    Age int
  )";
  mysql_query($sql,$con);

  mysql_close($con);
?>
```

在创建表之前，必须首先选择 MySQL 数据库，通过 mysql_select_db() 函数完成选取数据库操作。当创建 varchar 类型的数据库字段时，必须规定该字段的最大长度，例如 varchar(15)。

表 55.1: MySQL 数据类型

数据类型	描述
int(size) smallint(size) tinyint(size) mediumint(size) bigint(size)	仅支持整数。在 size 参数中规定数字的最大值。
decimal(size,d) double(size,d) float(size,d)	支持带有小数的数字。 在 size 参数中规定数字的最大值。在 d 参数中规定小数点右侧的数字的最大值。
char(size)	支持固定长度的字符串。(可包含字母、数字以及特殊符号)。 在 size 参数中规定固定长度。
varchar(size)	支持可变长度的字符串。(可包含字母、数字以及特殊符号)。 在 size 参数中规定最大长度。
tinytext	支持可变长度的字符串，最大长度是 255 个字符。

数据类型	描述
text blob	支持可变长度的字符串，最大长度是 65535 个字符。
mediumtext mediumblob	支持可变长度的字符串，最大长度是 16777215 个字符。
longtext longblob	支持可变长度的字符串，最大长度是 4294967295 个字符。
date(yyyy-mm-dd) datetime(yyyy-mm-dd hh:mm:ss) timestamp(yyyymmddhhmmss) time(hh:mm:ss)	支持日期或时间
enum(value1,value2,ect)	ENUM 是 ENUMERATED 列表的缩写。可以在括号中存放最多 65535 个值。
set	SET 与 ENUM 相似。但是 SET 可拥有最多 64 个列表项目，并可存放不止一个 choice

主键用于对表中的行进行唯一标识，每个表都应有一个主键字段。

每个主键值在表中必须是唯一的，而且主键字段不能为空，这是由于数据库引擎需要一个值来对记录进行定位。

下面的例子把 personID 字段设置为主键字段。主键字段通常是 ID 号，且通常使用 AUTO_INCREMENT 规定。AUTO_INCREMENT 会在新记录被添加时逐一增加该字段的值。

要确保主键字段不为空，就必须向该字段添加 NOT NULL 设置。

Example 40 主键

```
$sql = "CREATE TABLE Persons
(
  personID int NOT NULL AUTO_INCREMENT,
  PRIMARY KEY(personID),
  FirstName varchar(15),
  LastName varchar(15),
  Age int
)";

mysql_query($sql,$con);
```

主键字段永远要被编入索引，这条规则没有例外。

对主键字段进行索引才能使数据库引擎快速定位给予该键值的行。

INSERT INTO 语句用于向数据库表添加新记录。

```
INSERT INTO table_name
VALUES (value1, value2,....)
```

还可以规定希望在其中插入数据的列：

```
INSERT INTO table_name (column1, column2,...)
VALUES (value1, value2,....)
```

为了向 MySQL 数据库表插入数据，必须使用 mysql_query() 函数。下面的例子向 “Persons” 表添加了两个新记录：

Example 41 插入数据

```
<?php
 $con = mysql_connect("localhost","admin","testtest");
 if(!$con){
  die('Could not connect: ' . mysql_error() );
 }
 mysql_select("my_db",$con);
 mysql_query("INSERT INTO Persons(FirstName,LastName,Age)
 VALUES('Peter','Griffin','34')
 ");
 mysql_query("INSERT INTO Persons(FirstName,LastName,Age)
 VALUES('Green','Jim','30')
 ");

 mysql_close($con);
?>
```

把来自表单的数据插入数据库时，首先通过 $_POST 变量从表单取回值，然后 mysql_query() 函数执行 INSERT INTO 语句，这样一条新的记录才会添加到数据库表中。

Example 42 HTML 表单

```
<!DOCTYPE html>
<html>
<title>HTML Form Example</title>
<body>

<form action="insert.php" method="post">
Firstname: <input type="text" name="firstname" />
Lastname: <input type="text" name="lastname" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>

</body>
</html>
```

当用户点击示例中 HTML 表单中的提交按钮时，表单数据被发送到 “insert.php”，insert.php 中的脚本完成新数据插入操作。

Example 43　将 HTML 表单插入数据库

```
<?php
  $con = mysql_connect("localhost","admin","testtest");
  if(!$con){
    die('Could not connect: ' . mysql_error() );
  }
  mysql_select_db("my_db",$con);
  $sql = "INSERT INTO Persons(FirstName,LastName,Age)
  VALUES
  ('$_POST['firstname']','$_POST['lastname]','$_POST['age']')";
  if(!mysql_query($sql,$con)){
    die('Error: ' . mysql_error() );
  }
  echo "1 record added.";

  mysql_close($con);
?>
```

Example 44

Example 45

Example 46

Example 47

Part IX

Class and Objects

Chapter 56

Introduction

PHP 5 完全重写了对象模型以得到更佳性能和更多特性，从此 PHP 具备了完整的对象模型，现在 PHP 应用程序可以不再使用面向过程的方式进行开发，也不再需要按照代码书写的顺序基本一致的方向来从上而下执行。

PHP5 对待对象的方式与引用和句柄相同，即每个变量都持有对象的引用，而不是整个对象的拷贝。

另外，PHP 已经可以支持访问控制，抽象类和 final 类与方法，附加的魔术方法，接口，对象复制和类型约束等面向对象特性。

56.1 Class Definition

从结构体更进一步，类是一种封装了类对象的属性和方法的复合数据类型，因此创建一个类就相当于创造了一种新的数据类型。

每个类的定义都以关键字 class 开头，后面跟着类名，后面跟着一对花括号，里面包含有类的属性与方法的定义。

类名可以是任何非 PHP 保留字的合法标签。一个合法的类名以字母或下划线开头，后面跟着若干字母，数字或下划线。

以正则表达式表示类名为：`[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*`。

一个类可以包含有属于自己的常量、变量（称为“属性”）以及函数（称为“方法”）。

```
<?php
class SimpleClass
{
   // property declaration
   public $var = 'a default value';
```

```
    // method declaration
    public function displayVar() {
        echo $this->var;
    }
}
?>
```

当一个方法在类定义内部被调用时，有一个可用的伪变量 `$this`。

`$this` 是一个到主叫对象的引用（通常是该方法所从属的对象，但是如果是从第二个对象静态调用时也可能是另一个对象）。

下面是伪变量 $this 的一个示例。

```
<?php
class A
{
    function foo()
    {
        if (isset($this)) {
            echo '$this is defined (';
            echo get_class($this);
            echo ")\n";
        } else {
            echo "\$this is not defined.\n";
        }
    }
}

class B
{
    function bar()
    {
        // Note: the next line will issue a warning if E_STRICT is enabled.
        A::foo();
    }
}

$a = new A();
$a->foo();

// Note: the next line will issue a warning if E_STRICT is enabled.
A::foo();
$b = new B();
```

```
$b->bar();

// Note: the next line will issue a warning if E_STRICT is enabled.
B::bar();
?>
```

以上例程会输出：

```
$this is defined (A)
$this is not defined.
$this is defined (B)
$this is not defined.
```

56.2 Class Instance

作为一个复杂的数据类型，或者说一个对象的模板，类必须至少进行一次实例化才能使用。

要创建一个类的实例，必须使用 new 关键字。当创建新对象时该对象总是被赋值，除非该对象定义了构造函数并且在出错时抛出了一个异常。类应该在被实例化之前定义（某些情况下则必须这样）。

如果在 new 之后跟着的是一个包含有类名的字符串，则该类的一个实例被创建。如果该类属于一个名字空间，则必须使用其完整名称。

```
<?php
$instance = new SimpleClass();

// 也可以这样做:
$className = 'Foo';
$instance = new $className(); // Foo()
?>
```

在类定义内部，可以用 `new self` 和 `new parent` 创建新对象。

当把一个对象已经创建的实例赋给一个新变量时，新变量会访问同一个实例，就和用该对象赋值一样。此行为和向函数传递入实例时一样。

```
<?php

$instance = new SimpleClass();

$assigned =  $instance;
$reference =& $instance;
```

```
$instance->var = '$assigned will have this value';

$instance = null; // $instance and $reference become null

var_dump($instance);
var_dump($reference);
var_dump($assigned);
?>
```

以上例程会输出：

```
NULL
NULL
object(SimpleClass)#1 (1) {
   ["var"]=>
     string(30) "$assigned will have this value"
}
```

除了可以用克隆给一个已创建的对象建立一个新实例之外，PHP 还引入了两个新方法来创建一个对象的实例，例如：

```
<?php
class Test
{
   static public function getNew()
   {
      return new static;
   }
}

class Child extends Test
{}

$obj1 = new Test();
$obj2 = new $obj1;
var_dump($obj1 !== $obj2);

$obj3 = Test::getNew();
var_dump($obj3 instanceof Test);

$obj4 = Child::getNew();
```

```
var_dump($obj4 instanceof Child);
?>
```

以上例程会输出：

```
bool(true)
bool(true)
bool(true)
```

除了使用 new 关键字来实例化类之外，还可以使用构造函数来进行对象初始化。

- 构造函数和类同名；
- 构造函数用于初始化类的对象实例；
- 构造函数没有返回值。

```
<?php
class Employee
{
  var $id;
  var $name;
  var $birthday;
  var $salary;

  function setId($id){
    $this->id = $id;
  }
  function setName($name){
    $this->name = strtoupper($name);
  }
  function setBirthday($birthday){
    $this->birthday = $birthday;
  }
  function setSalary($salary){
    $this->salary = $salary;
  }

  function getId(){
    return $this->id;
  }
  function getName(){
    return $this->name;
  }
  function getBirthday(){
```

```
    return $this->birthday;
  }
  function getSalary(){
    return $this->salary;
  }

  function getEmployeeInfo(){
    $info = "员工号: " . $this->getId() . " | ";
    $info .= "姓名: " . $this->getName() . " | ";
    $info .= "生日: " . $this->getBirthday() . " | ";
    $info .= "工资: " . $this->getSalary() . "<br />";

    return $info;
  }

  function Employee($id, $name, $birthday, $salary){
    $this->setId($id);
    $this->setName($name);
    $this->setBirthday($birthday);
    $this->setSalary($salary);
  }
}
?>
```

56.3 Class Inheritance

一个类可以在声明中用 extends 关键字继承另一个类的方法和属性，但是父类不能调用子类的任何方法。

PHP 不支持多重继承，一个类只能继承一个基类。

```
<?php
class ProjectManager extends Employee
{
  var $project_name;
  var $sub_employees = array();

  function ProjectManager($id, $name, $birthday, $salary){
    $this->Employee($id, $name, $birthday, $salary);
  }

  function setProjectName($pjName){
```

```
    $this->project_name = $pjName;
  }
  function getProjectName(){
    return $this->project_name;
  }

  function addEmployee(&$employee){
    $this->sub_employee[] = &$employee;
  }
}
?>
```

被继承的方法和属性可以被同样的名字重新声明来覆盖（overriding）。

如果父类定义方法时使用了 final，则该方法不可被覆盖，可以通过 parent:: 来访问被覆盖的方法或属性。

当覆盖方法时，参数必须保持一致否则 PHP 将发出 E_STRICT 级别的错误信息。只有构造函数例外，构造函数可在被覆盖时使用不同的参数。

```
<?php
class ExtendClass extends SimpleClass
{
    // Redefine the parent method
    function displayVar()
    {
        echo "Extending class\n";
        parent::displayVar();
    }
}

$extended = new ExtendClass();
$extended->displayVar();
?>
```

以上例程会输出：

```
Extending class
a default value
```

56.4 Class Resolution

PHP 的关键词 class 也可用于类名的解析，使用 ClassName::class 就可以获取一个包含了类 ClassName 的完全限定名称的字符串，对于使用了命名空间的类尤其有用。

```
<?php
namespace NS {
    class ClassName {
    }

    echo ClassName::class;
}
?>
```

以上例程会输出：

```
NS\ClassName
```

Chapter 57

Class Attributes

类的成员变量统一叫做“属性”(或者叫“字段”、“特征”)，可以是基本的数据类型，也可以是其他对象。

属性声明是由关键字 public，protected 或者 private 开头，然后跟一个普通的变量声明来组成。

属性中的变量可以初始化，但是初始化的值必须是常数，这里的常数是指 PHP 脚本在编译阶段时就可以得到其值，而不依赖于运行时的信息才能求值。

```
<?php
class SimpleClass
{
   // 错误的属性声明
   public $var1 = 'hello ' . 'world';
   public $var2 = <<<EOD
hello world
EOD;
   public $var3 = 1+2;
   public $var4 = self::myStaticMethod();
   public $var5 = $myVar;

   // 正确的属性声明
   public $var6 = myConstant;
   public $var7 = array(true, false);

   //在 PHP 5.3.0 及之后，下面的声明也正确
   public $var8 = <<<'EOD'
hello world
EOD;
```

```
}
?>
```

PHP 5 声明属性依然可以直接使用关键字 var 来替代（或者附加于）public，protected 或 private，只是 var 不再是必要的。

另外，如果直接使用 var 声明属性，而没有用 public，protected 或 private 之一，PHP 5 会将其视为 public。

跟 heredocs 不同，nowdocs 可在任何静态数据上下文中使用，包括属性声明。

```
<?php
class foo {
  public $bar = <<<'EOT'
bar
EOT;
}
?>
```

Chapter 58

Class Methods

类的方法也称为类的成员方法，其作用相当于函数，方法只能在类的内部定义。

```
<?php
class Employee
{
  var $id;
  var $name;
  var $birthday;
  var $salary;

  function setId($id){
    $this->id = $id;
  }
  function setName($name){
    $this->name = strtoupper($name);
  }
  function setBirthday($birthday){
    $this->birthday = $birthday;
  }
  function setSalary($salary){
    $this->salary = $salary;
  }

  function getId(){
    return $this->id;
  }
  function getName(){
    return $this->name;
```

```
  }
  function getBirthday(){
    return $this->birthday;
  }
  function getSalary(){
    return $this->salary;
  }

  function getEmployeeInfo(){
    $info = "员工号: " . $this->getId() . " | ";
    $info .= "姓名: " . $this->getName() . " | ";
    $info .= "生日: " . $this->getBirthday() . " | ";
    $info .= "工资: " . $this->getSalary() . "<br />";

    return $info;
  }
}
?>
```

在类的成员方法里面，可以用 ->（对象运算符）：$this->property 的方式来访问非静态属性，静态属性则使用::（类似 self::$property）来访问。

当一个方法在类定义内部被调用时，都有一个可用的伪变量 $this。$this 是一个到主叫对象的引用（通常是该方法所从属的对象，但是如果是从第二个对象静态调用时也可能是另一个对象）。

Chapter 59

Class Constants

可以把在类或接口中始终保持不变的值定义为常量，而且在定义和使用常量的时候不需要使用 $ 符号。

常量的值必须是一个定值，不能是变量、类属性或数学运算的结果或函数调用。

PHP 允许用一个变量来动态调用类，但是该变量的值不能为关键字（如 self，parent 或 static）。

```
<?php
class MyClass
{
    const constant = 'constant value';

    function showConstant() {
        echo self::constant . "\n";
    }
}

echo MyClass::constant . "\n";

$classname = "MyClass";
echo $classname::constant . "\n"; // 自 5.3.0 起

$class = new MyClass();
$class->showConstant();

echo $class::constant."\n"; // 自 PHP 5.3.0 起
?>
```

和 heredoc 不同，nowdoc 可以用在任何静态数据中。

```
<?php
class foo {
    // 自 PHP 5.3.0 起
    const bar = <<<'EOT'
bar
EOT;
}
?>
```

Chapter 60

Class Autoload

在 PHP5 之前，编写面向对象的应用程序时需要对每个类的定义建立一个 PHP 源文件，而且往往不得不在每个 PHP 程序脚本开头引入一个长长的包含文件列表 (每个类一个文件)。

PHP5 取消了上述限制，可以定义一个 __autoload() 函数，它会在试图使用尚未被定义的类时被自动调用。

PHP 脚本引擎通过调用 __autoload() 函数可以实现在 PHP 出错失败前获得最后一个机会加载所需的类，只是自动加载不可用于 PHP 的 CLI 交互模式。

现在，spl_autoload_register() 提供了一种更加灵活的方式来实现类的自动加载，因此 __autoload() 函数已经不再建议使用，在以后的版本中可能废弃 __autoload() 函数。

- 在 PHP 5.3.0 之前，__autoload 函数抛出的异常不能被 catch 语句块捕获并会导致一个致命错误。
- 从 PHP 5.3.0+ 开始，__autoload 函数抛出的异常可以被 catch 语句块捕获，但是需要遵循一个条件。如果抛出的是一个自定义异常，那么必须存在相应的自定义异常类。__autoload 函数可以递归的自动加载自定义异常类。

如果类名被用于 call_user_func() 等环境时，它可能包含一些危险的字符（比如 ../），建议在这样的函数中不要使用用户的输入，起码需要在 __autoload() 时对输入进行验证。

本例尝试分别从 MyClass1.php 和 MyClass2.php 文件中加载 MyClass1 和 MyClass2 类。

```
<?php
function __autoload($class_name) {
   require_once $class_name . '.php';
}

$obj = new MyClass1();
$obj2 = new MyClass2();
?>
```

本例尝试加载接口 ITest。

```
<?php

function __autoload($name) {
    var_dump($name);
}

class Foo implements ITest {
}

/*
string(5) "ITest"

Fatal error: Interface 'ITest' not found in ...
*/
?>
```

本例抛出一个异常并在 try/catch 语句块中演示。

```
<?php
function __autoload($name) {
    echo "Want to load $name.\n";
    throw new Exception("Unable to load $name.");
}

try {
    $obj = new NonLoadableClass();
} catch (Exception $e) {
    echo $e->getMessage(), "\n";
}
?>
```

以上例程会输出：

```
Want to load NonLoadableClass.
Unable to load NonLoadableClass.
```

本例将一个异常抛给不存在的自定义异常处理函数。

```
<?php
function __autoload($name) {
    echo "Want to load $name.\n";
    throw new MissingException("Unable to load $name.");
```

```
}

try {
    $obj = new NonLoadableClass();
} catch (Exception $e) {
    echo $e->getMessage(), "\n";
}
?>
```

以上例程会输出:

```
Want to load NonLoadableClass.
Want to load MissingException.

Fatal error: Class 'MissingException' not found in testMissingException.php on line 4
```

Chapter 61

Constructor

```
void __construct ([ mixed $args [, $... ]] )
```

PHP 允行用户在一个类中定义一个方法作为构造函数，具有构造函数的类会在每次创建新对象时先调用此方法，所以构造函数非常适合在使用对象之前执行一些初始化操作。

- 如果子类中定义了构造函数，则不会隐式调用其父类的构造函数。
- 如果需要显式地执行父类的构造函数，需要在子类的构造函数中调用 parent::__construct()。
- 如果子类没有定义构造函数则会如同一个普通的类方法一样从父类继承构造函数（假如没有被定义为 private 的话）。

```
<?php
class BaseClass {
   function __construct() {
       print "In BaseClass constructor\n";
   }
}

class SubClass extends BaseClass {
   function __construct() {
       parent::__construct();
       print "In SubClass constructor\n";
   }
}

class OtherSubClass extends BaseClass {
    // inherits BaseClass's constructor
}

// In BaseClass constructor
```

```
$obj = new BaseClass();

// In BaseClass constructor
// In SubClass constructor
$obj = new SubClass();

// In BaseClass constructor
$obj = new OtherSubClass();
?>
```

为了实现向后兼容性，如果 PHP 5 在类中找不到 __construct() 函数并且也没有从父类继承一个的话，它就会尝试寻找旧式的构造函数，也就是和类同名的函数，因此唯一会产生兼容性问题的情况就是类中已有一个名为 __construct() 的方法却被用于其它用途。

与其它方法不同，当 __construct() 被与父类 __construct() 具有不同参数的方法覆盖时，PHP 不会产生一个 E_STRICT 错误信息。

在命名空间中，与类名同名的方法不再作为构造函数，而且这一改变不影响不在命名空间中的类。

```
<?php
namespace Foo;
class Bar {
   public function Bar() {
      // treated as constructor in PHP 5.3.0-5.3.2
      // treated as regular method as of PHP 5.3.3
   }
}
?>
```

Chapter 62

Destructor

```
void __destruct ( void )
```

PHP 的析构函数的概念类似于其它面向对象的语言（例如 C++），析构函数会在到某个对象的所有引用都被删除或者当对象被显式销毁时执行。

```
<?php
class MyDestructableClass {
   function __construct() {
      print "In constructor\n";
      $this->name = "MyDestructableClass";
   }

   function __destruct() {
      print "Destroying " . $this->name . "\n";
   }
}

$obj = new MyDestructableClass();
?>
```

和构造函数一样，父类的析构函数不会被 PHP 引擎隐式调用。

如果要显式地执行父类的析构函数，必须在子类的析构函数体中显式调用 parent::__destruct()。

和构造函数一样，子类如果自己没有定义析构函数则默认会继承父类的。

即使在使用 exit() 终止脚本运行时，析构函数也会被调用。在析构函数中调用 exit() 将会中止其余关闭操作的运行。

试图在析构函数（在脚本终止时被调用）中抛出一个异常会导致致命错误。

析构函数在脚本关闭时调用，此时所有的 HTTP 头信息已经发出，而且脚本关闭时的工作目录有可能和在 SAPI（如 apache）中时不同。

Chapter 63

Visibility

对属性或方法的访问控制是通过关键字 public（公有），protected（受保护）或 private（私有）来实现的。

- 被定义为公有的类成员可以在任何地方被访问。
- 被定义为受保护的类成员则可以被其自身以及其子类和父类访问。
- 被定义为私有的类成员则只能被其定义所在的类访问。

63.1 Property Visibility

类属性必须定义为公有，受保护，私有之一（如果用 var 定义，则被视为公有）。

```
<?php
/**
 * Define MyClass
 */
class MyClass
{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}
```

```
$obj = new MyClass();
echo $obj->public; // 这行能被正常执行
echo $obj->protected; // 这行会产生一个致命错误
echo $obj->private; // 这行也会产生一个致命错误
$obj->printHello(); // 输出 Public、Protected 和 Private


/**
 * Define MyClass2
 */
class MyClass2 extends MyClass
{
    // 可以对 public 和 protected 进行重定义，但 private 而不能
    protected $protected = 'Protected2';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj2 = new MyClass2();
echo $obj2->public; // 这行能被正常执行
echo $obj2->private; // 未定义 private
echo $obj2->protected; // 这行会产生一个致命错误
$obj2->printHello(); // 输出 Public、Protected2 和 Undefined

?>
```

为了兼容性考虑，在 PHP 4 中使用 var 关键字对变量进行定义的方法在 PHP 5 中仍然有效，只是已经成为了 public 关键字的一个别名。

63.2 Method Visibility

类中的方法可以被定义为公有，私有或受保护。如果没有设置这些关键字，则该方法默认为公有。

```
<?php
```

```
/**
 * Define MyClass
 */
class MyClass
{
    // 声明一个公有的构造函数
    public function __construct() { }

    // 声明一个公有的方法
    public function MyPublic() { }

    // 声明一个受保护的方法
    protected function MyProtected() { }

    // 声明一个私有的方法
    private function MyPrivate() { }

    // 此方法为公有
    function Foo()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate();
    }
}

$myclass = new MyClass;
$myclass->MyPublic(); // 这行能被正常执行
$myclass->MyProtected(); // 这行会产生一个致命错误
$myclass->MyPrivate(); // 这行会产生一个致命错误
$myclass->Foo(); // 公有，受保护，私有都可以执行


/**
 * Define MyClass2
 */
class MyClass2 extends MyClass
{
    // 此方法为公有
    function Foo2()
    {
```

```
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate(); // 这行会产生一个致命错误
    }
}

$myclass2 = new MyClass2;
$myclass2->MyPublic(); // 这行能被正常执行
$myclass2->Foo2(); // 公有的和受保护的都可执行，但私有的不行

class Bar
{
    public function test() {
        $this->testPrivate();
        $this->testPublic();
    }

    public function testPublic() {
        echo "Bar::testPublic\n";
    }

    private function testPrivate() {
        echo "Bar::testPrivate\n";
    }
}

class Foo extends Bar
{
    public function testPublic() {
        echo "Foo::testPublic\n";
    }

    private function testPrivate() {
        echo "Foo::testPrivate\n";
    }
}

$myFoo = new foo();
$myFoo->test(); // Bar::testPrivate
                // Foo::testPublic
?>
```

63.3 Object Visibility

同一个类的对象即使不是同一个实例也可以互相访问对方的私有与受保护成员。这是由于在这些对象的内部具体实现的细节都是已知的。

```
<?php
class Test
{
    private $foo;

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

    private function bar()
    {
        echo 'Accessed the private method.';
    }

    public function baz(Test $other)
    {
        // We can change the private property:
        $other->foo = 'hello';
        var_dump($other->foo);

        // We can also call the private method:
        $other->bar();
    }
}

$test = new Test('test');

$test->baz(new Test('other'));
?>
```

以上例程会输出：

```
string(5) "hello"
Accessed the private method.
```

Chapter 64

Inheritance

继承会影响类与类，对象与对象之间的关系。例如，当扩展一个类时，子类就会继承父类所有公有的和受保护的方法。

除非子类覆盖了父类的方法，被继承的方法都会保留其原有功能，因此继承对于功能的设计和抽象是非常有用的，而且类似的对象增加新功能无须重新重复这些公用的功能。

```
<?php
class foo
{
    public function printItem($string)
    {
        echo 'Foo: ' . $string . PHP_EOL;
    }

    public function printPHP()
    {
        echo 'PHP is great.' . PHP_EOL;
    }
}

class bar extends foo
{
    public function printItem($string)
    {
        echo 'Bar: ' . $string . PHP_EOL;
    }
}

$foo = new foo();
```

```
$bar = new bar();
$foo->printItem('baz'); // Output: 'Foo: baz'
$foo->printPHP();   // Output: 'PHP is great'
$bar->printItem('baz'); // Output: 'Bar: baz'
$bar->printPHP();   // Output: 'PHP is great'
?>
```

除非使用了自动加载，否则一个类必须在使用之前被定义。如果一个类扩展了另一个，则父类必须在子类之前被声明，同样适用于类继承其它类与接口。

Chapter 65

Scope Resolution

范围解析操作符（也可称作 Paamayim Nekudotayim）可以用于访问静态成员和类常量，还可以用于覆盖类中的属性和方法。

当在类定义之外引用到这些项目时，要使用类名。

PHP 允许通过变量来引用类，该变量的值不能是关键字（例如 self，parent 和 static）。

```
<?php
class MyClass {
    const CONST_VALUE = 'A constant value';
}

$classname = 'MyClass';
echo $classname::CONST_VALUE; // 自 PHP 5.3.0 起

echo MyClass::CONST_VALUE;
?>
```

self，parent 和 static 这三个特殊的关键字是用于在类定义的内部对其属性或方法进行访问的。

```
<?php
class OtherClass extends MyClass
{
    public static $my_static = 'static var';

    public static function doubleColon() {
        echo parent::CONST_VALUE . "\n";
        echo self::$my_static . "\n";
    }
}
```

```
$classname = 'OtherClass';
echo $classname::doubleColon(); // 自 PHP 5.3.0 起

OtherClass::doubleColon();
?>
```

当一个子类覆盖其父类中的方法时，PHP 不会调用父类中已被覆盖的方法，是否调用父类的方法取决于子类。这种机制也适用于构造函数、析构函数、重载以及魔术方法。

```
<?php
class MyClass
{
    protected function myFunc() {
        echo "MyClass::myFunc()\n";
    }
}

class OtherClass extends MyClass
{
    // 覆盖了父类的定义
    public function myFunc()
    {
        // 但还是可以调用父类中被覆盖的方法
        parent::myFunc();
        echo "OtherClass::myFunc()\n";
    }
}

$class = new OtherClass();
$class->myFunc();
?>
```

Chapter 66

Static Keyword

static 关键字可以用来定义静态方法和属性，也可用于定义静态变量以及后期静态绑定。

声明类属性或方法为静态，就可以不实例化类而直接访问，不过用静态方式调用一个非静态方法会导致一个 E_STRICT 级别的错误。

静态属性不能通过一个类已实例化的对象来访问（但是静态方法可以）。

为了兼容 PHP 4，如果没有指定访问控制，属性和方法默认为公有。

- 静态方法不需要通过对象即可调用会导致伪变量 $this 在静态方法中不可用。
- 静态属性不可以由对象通过 -> 操作符来访问。

和所有静态变量一样，静态属性只能被初始化为字符串或常量，不能使用表达式，因此可以把静态属性初始化为整数或数组，但是不能初始化为另一个变量或函数返回值，也不能指向一个对象。

虽然可以用一个变量来动态调用类，但是该变量的值不能为关键字 self，parent 或 static。

```
<?php
class Foo
{
   public static $my_static = 'foo';

   public function staticValue() {
       return self::$my_static;
   }
}

class Bar extends Foo
{
   public function fooStatic() {
       return parent::$my_static;
```

```
    }
}


print Foo::$my_static . "\n";

$foo = new Foo();
print $foo->staticValue() . "\n";
print $foo->my_static . "\n"; // Undefined "Property" my_static

print $foo::$my_static . "\n";
$classname = 'Foo';
print $classname::$my_static . "\n"; // As of PHP 5.3.0

print Bar::$my_static . "\n";
$bar = new Bar();
print $bar->fooStatic() . "\n";
?>
```

Chapter 67

Class Abstraction

PHP 支持抽象类和抽象方法，抽象类不能被直接实例化。

67.1 Abstract class

具体来说，抽象类是指在 class 前加了 abstract 关键字且存在抽象方法（在类方法 function 关键字前加了 abstract 关键字）的类。

- 任何一个类，如果它里面至少有一个方法是被声明为抽象的，那么这个类就必须被声明为抽象的。
- 被定义为抽象的方法只是声明了其调用方式（参数），不能定义其具体的功能实现。
- 继承一个抽象类的时候，子类必须实现父类中的所有抽象方法，这些方法的访问控制必须和父类中一样（或者更为宽松）。

例如，某个抽象方法被声明为受保护的，那么子类中实现的方法就应该声明为受保护的或者公有的，而不能定义为私有的。

如果 B 实现了抽象类 A 的抽象方法 abstract_func() ，那么 B 中 abstract_func() 方法的访问控制不能比 A 中 abstract_func() 的访问控制更严格，也就是说：

1. 如果 A 中 abstract_func() 声明为 public ，那么 B 中 abstract_func() 的声明只能是 public ，不能是 protected 或 private。
2. 如果 A 中 abstract_func() 声明为 protected ，那么 B 中 abstract_func() 的声明可以是 public 或 protected ，但不能是 private。
3. 抽象类不能指定 abstract_func() 声明为 private，否则产生 Fatal error : Abstract function A::abstract_func() cannot be declared private。

抽象类中只定义（或部分实现）子类需要的方法。子类可以通过继承抽象类并通过实现抽象类中的所有抽象方法，使抽象类具体化。

另外，方法的调用方式必须匹配，即类型和所需参数数量必须一致。例如，子类定义了一个可选参数，而父类抽象方法的声明里没有，则两者的声明并无冲突。

上述规则同样适用于构造函数。

Example 48 抽象类示例

```
<?php
abstract class AbstractClass
{
 // 强制要求子类定义这些方法
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    // 普通方法（非抽象方法）
    public function printOut() {
        print $this->getValue() . "\n";
    }
}

class ConcreteClass1 extends AbstractClass
{
    protected function getValue() {
        return "ConcreteClass1";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}

class ConcreteClass2 extends AbstractClass
{
    public function getValue() {
        return "ConcreteClass2";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass2";
    }
}

$class1 = new ConcreteClass1;
```

```
$class1->printOut();
echo $class1->prefixValue('FOO_') ."\n";

$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') ."\n";
?>
```

以上例程会输出：

```
ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2
```

如果子类需要实例化，前提是它实现了抽象类中的所有抽象方法。如果子类没有全部实现抽象类中的所有抽象方法，那么该子类也是一个抽象类，必须在 class 前面加上 abstract 关键字，并且不能被实例化。

Example 49　子类可以定义父类签名中不存在的可选参数

```
<?php
abstract class AbstractClass
{
    // 抽象方法仅需要定义需要的参数
    abstract protected function prefixName($name);

}

class ConcreteClass extends AbstractClass
{

    // 子类可以定义父类签名中不存在的可选参数
    public function prefixName($name, $separator = ".") {
        if ($name == "Pacman") {
            $prefix = "Mr";
        } elseif ($name == "Pacwoman") {
            $prefix = "Mrs";
        } else {
            $prefix = "";
        }
```

```
        return "{$prefix}{$separator} {$name}";
    }
}

$class = new ConcreteClass;
echo $class->prefixName("Pacman"), "\n";
echo $class->prefixName("Pacwoman"), "\n";
?>
```

以上例程会输出：

```
Mr. Pacman
Mrs. Pacwoman
```

抽象类主要应用在既需要统一的接口，又需要实例变量或缺省的方法的情况中，其具体应用场合包括如下：

1. 定义了一组接口，但又不想强迫每个实现类都必须实现所有的接口。可以用 abstract class 定义一组方法体（可以是空方法体），然后由子类选择自己所感兴趣的方法来覆盖。
2. 某些场合下，只靠纯粹的接口不能满足类与类之间的协调，还必须使用类中表示状态的变量来区别不同的关系。abstract 的中介作用可以很好地满足这一点。
3. 规范了一组相互协调的方法，其中一些方法是共同的，与状态无关的，可以共享的，无需子类分别实现，而另一些方法却需要各个子类根据自己特定的状态来实现特定的功能。

旧代码中如果没有自定义类或函数被命名为“abstract”，则可以不加修改地正常运行。

67.2 Abstract mode

事实上，对于 PHP 编程来说，抽象类可以实现的功能，接口也可以实现。

在实际应用中，抽象类和接口的区别，不在于编程实现，而在于程序设计模式的不同。

一般来讲，抽象用于不同的事物，接口则用于事物的行为。例如，水生生物是鲸鱼的抽象概念，但是水生生物并不是鲸鱼的行为，吃东西才是鲸鱼的行为。

对于大型项目来说，对象都是由基本的抽象类继承实现，而这些类的方法通常都由接口来定义。

此外，对于事物属性的更改，建议使用接口，而不是直接赋值或者其他的方式。

```
<?php
interface IAction {
 public function eat();
```

```
}
class Whale implements IAction {
  public function eat() {
    echo "Whale eat fish.\n";
  }
}
class Carp implements IAction {
  public function eat() {
    echo "Carp eat moss.\n";
  }
}

class Observer {
  public function __construct() {
    $whale = new Whale();
    $carp = new Carp();
    $this->observeEat($whale);
    $this->observeEat($carp);
  }
  function observeEat(IAction $animal) {
    $animal->eat();
  }
}
$observer = new observer();
?>
```

运行结果如下：

```
$ php Observer.php
Whale eat fish.
Carp eat moss.
```

好的设计模式是严格对问题进行抽象，虽然抽象类和接口对于编程实现来说是类似的，但是对于程序设计模式是不同的。

Chapter 68

Object Interfaces

接口（interface）可以用于指定某个类必须实现哪些方法，但是不需要定义这些方法的具体内容。

接口是通过 interface 关键字来定义的，就像定义一个标准的类一样。

- 接口中定义的所有的方法必须都是空的。
- 接口中定义的所有的方法必须都是公有的，这是接口的特性。
- 接口也可以继承，通过使用 extends 操作符。

和抽象类相比，接口是完全抽象的，只能声明方法，而且只能声明 public 的方法，不能声明 private 及 protected 的方法，不能定义方法体，也不能声明实例变量。

抽象类提供了具体实现的标准，而接口则是纯粹的模版，在接口中只定义功能，而不包含实现的内容。

虽然接口也可以声明常量变量，但是将常量变量放在 interface 中违背了其作为接口的作用而存在的宗旨，也混淆了 interface 与类的不同价值。如果的确需要，可以将其放在相应的 abstract class 或 class 中。

68.1 Implements

要实现一个接口，使用 implements 操作符。

- 类中必须实现接口中定义的所有方法，否则会报告致命错误。
- 类可以实现多个接口，用逗号来分隔多个接口的名称。
- 实现多个接口时，接口中的方法不能有重名。

类要实现接口，必须使用和接口中所定义的方法完全一致的方式，否则会导致致命错误。

```
<?php
// 声明一个'iTemplate'接口
```

```
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}


// 实现接口
// 下面的写法是正确的
class Template implements iTemplate
{
    private $vars = array();

    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }

    public function getHtml($template)
    {
        foreach($this->vars as $name => $value) {
            $template = str_replace('{' . $name . '}', $value, $template);
        }

        return $template;
    }
}

// 下面的写法是错误的，因为没有实现 getHtml():
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
    private $vars = array();

    public function setVariable($name, $var)
    {
        $this->vars[$name] = $var;
    }
}
?>
```

实现接口的具体过程和继承一个仅包含抽象方法的抽象类是一样的。一个类可以同时继承一个父类和实现任意多个接口。extends 子句应该在 implements 子句之前。

PHP 只支持继承自一个父类，因此 extends 关键字后只能跟一个类名。

Example 50　可扩充的接口

```
<?php
interface a
{
    public function foo();
}

interface b extends a
{
    public function baz(Baz $baz);
}

// 正确写法
class c implements b
{
    public function foo()
    {
    }

    public function baz(Baz $baz)
    {
    }
}

// 错误写法会导致一个致命错误
class d implements b
{
    public function foo()
    {
    }

    public function baz(Foo $foo)
    {
    }
}
?>
```

接口不可以实现另一个接口，但是可以继承多个接口。

Example 51 继承多个接口

```
<?php
interface a
{
    public function foo();
}

interface b
{
    public function bar();
}

interface c extends a, b
{
    public function baz();
}

class d implements c
{
    public function foo()
    {
    }

    public function bar()
    {
    }

    public function baz()
    {
    }
}
?>
```

接口的应用场合包括如下：

1. 类与类之间需要特定的接口进行协调，而不关心其如何实现。
2. 作为能够实现特定功能的标识存在，也可以是没有接口方法的纯粹标识。
3. 需要将一组类视为单一的类，而调用者只通过接口来与这组类发生联系。
4. 需要实现特定的多项功能，而这些功能之间可能完全没有任何联系。

68.2 Constants

接口中也可以定义常量，接口常量和类常量的使用完全相同，但是不能被子类或子接口所覆盖。

Example 52 使用接口常量

```php
<?php
interface a
{
    const b = 'Interface constant';
}

// 输出接口常量
echo a::b;

// 错误写法，因为常量不能被覆盖。接口常量的概念和类常量是一样的。
class b implements a
{
    const b = 'Class constant';
}
?>
```

接口加上类型约束提供了一种良好的方式来确保某个对象包含有某些方法。

68.3 Abstract

抽象类提供了具体实现的标准，而接口则是纯粹的模版。

- 接口只定义功能，而不包含实现的内容。
- 抽象类只定义（或部分实现）子类需要的方法。

interface 强调特定功能的实现，而 abstract class 强调所属关系。

尽管 interface 实现类及 abstract class 的子类都必须要实现相应的抽象方法，但实现的形式不同。

- interface 中的每一个方法都是抽象方法，都只是声明的 (没有方法体)，实现类必须要实现。
- abstract class 的子类可以有选择地实现，abstract class 中并非所有的方法都是抽象的，只有声明为 abstract 的方法才是抽象的，子类必须实现，没有 abstract 的方法在 abstract class 中必须定义方法体。

- abstract class 的子类在继承它时，对非抽象方法既可以直接继承，也可以覆盖。对抽象方法，可以选择实现，也可以留给其子类来实现，但是这个类必须也声明为抽象类，因此也不能实例化。

abstract class 是 interface 与 class 的中介。abstract class 在 interface 及 class 中起到了承上启下的作用。一方面 abstract class 是抽象的，可以声明抽象方法来规范子类必须实现的功能，另一方面 abstract class 又可以定义缺省的方法体来让子类直接使用或覆盖，最后 abstract class 还可以定义自己的实例变量来让子类通过继承来使用。

接口中的抽象方法前不用也不能加 abstract 关键字，默认隐式就是抽象方法，也不能加 final 关键字来防止抽象方法的继承，但是抽象类中抽象方法前则必须加上 abstract 表示显示声明为抽象方法。

接口中的抽象方法默认是 public 的，也只能是 public 的，不能用 private ，protected 修饰符修饰。而抽象类中的抽象方法则可以用 public ，protected 来修饰，不允许用 private 。

Chapter 69

Predefined Interfaces

69.1 Traversable interface

Traversable interface 是检测一个类是否可以使用 foreach 进行遍历的接口，该接口是无法被单独实现的基本抽象接口。

这是一个无法在 PHP 脚本中实现的内部引擎接口。IteratorAggregate 或 Iterator 接口可以用来代替它。

Traversable interface 必须由 IteratorAggregate 或 Iterator 接口实现，但实现此接口的内建类可以使用 foreach 进行遍历而无需实现 IteratorAggregate 或 Iterator 接口。

```
Traversable {
}
```

这个接口没有任何方法，它的作用仅仅是作为所有可遍历类的基本接口。

While you cannot implement this interface, you can use it in your checks to determine if something is usable in for each. Here is what I use if I'm expecting something that must be iterable via foreach.

```
<?php
   if( !is_array( $items ) && !$items instanceof Traversable )
      //Throw exception here
?>
```

69.2 Iterator interface

Iterator interface 是可在内部迭代自己的外部迭代器或类的接口。

```
Iterator extends Traversable {
```

```
  /* 方法 */
  abstract public mixed current ( void )
  abstract public scalar key ( void )
  abstract public void next ( void )
  abstract public void rewind ( void )
  abstract public boolean valid ( void )
}
```

Iterator interface **方法**

- Iterator::current —返回当前元素
- Iterator::key —返回当前元素的键
- Iterator::next —向前移动到下一个元素
- Iterator::rewind —返回到迭代器的第一个元素
- Iterator::valid —检查当前位置是否有效

PHP 已经提供了一些用于日常任务的迭代器。

Example 53 Iterator Interface 基本用法

```
<?php
class myIterator implements Iterator {
    private $position = 0;
    private $array = array(
        "firstelement",
        "secondelement",
        "lastelement",
    );

    public function __construct() {
        $this->position = 0;
    }

    function rewind() {
        var_dump(__METHOD__);
        $this->position = 0;
    }

    function current() {
        var_dump(__METHOD__);
        return $this->array[$this->position];
    }

    function key() {
```

```
        var_dump(__METHOD__);
        return $this->position;
    }

    function next() {
        var_dump(__METHOD__);
        ++$this->position;
    }

    function valid() {
        var_dump(__METHOD__);
        return isset($this->array[$this->position]);
    }
}

$it = new myIterator;

foreach($it as $key => $value) {
    var_dump($key, $value);
    echo "\n";
}
?>
```

这个例子展示了使用 foreach 时，迭代器方法的调用顺序。

以上例程的输出类似于：

```
string(18) "myIterator::rewind"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(0)
string(12) "firstelement"

string(16) "myIterator::next"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(1)
string(13) "secondelement"
```

```
string(16) "myIterator::next"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(2)
string(11) "lastelement"

string(16) "myIterator::next"
string(17) "myIterator::valid"
```

Iterator::current

Iterator::current —返回当前元素，此函数没有参数，可返回任何类型。

```
abstract public mixed Iterator::current ( void )
```

Iterator::key

Iterator::key —返回当前元素的键，此函数没有参数。成功返回标量，失败则返回 NULL 并分发 E_NOTICE 级错误。

```
abstract public scalar Iterator::key ( void )
```

Iterator::next

Iterator::next —移动当前位置到下一个元素。此函数没有参数，并且任何值返回都将被忽略。

```
abstract public void Iterator::next ( void )
```

此方法在 foreach 循环之后被调用。

Iterator::rewind

Iterator::rewind —返回到迭代器的第一个元素，此函数没有参数，并且任何值返回都将被忽略。

```
abstract public void Iterator::rewind ( void )
```

当开始一个 foreach 循环时，这是第一个被调用的方法。它将不会在 foreach 循环之后被调用。

Iterator::valid

Iterator::valid —检查当前位置是否有效，此函数没有参数。返回至将被转换为布尔类型，成功时返回 TRUE，或者在失败时返回 FALSE。

```
abstract public boolean Iterator::valid ( void )
```

此方法在 Iterator::rewind() 和 Iterator::next() 方法之后被调用以此用来检查当前位置是否有效。

69.3 IteratorAggregate interface

IteratorAggregate interface ——创建外部迭代器的接口。

```
IteratorAggregate extends Traversable {
/* 方法 */
abstract public Traversable getIterator ( void )
}
```

IteratorAggregate **方法**

- IteratorAggregate::getIterator —获取一个外部迭代器

Example 54 基本用法

```
<?php
class myData implements IteratorAggregate {
    public $property1 = "Public property one";
    public $property2 = "Public property two";
    public $property3 = "Public property three";

    public function __construct() {
        $this->property4 = "last property";
    }

    public function getIterator() {
        return new ArrayIterator($this);
    }
}

$obj = new myData;

foreach($obj as $key => $value) {
    var_dump($key, $value);
    echo "\n";
}
?>
```

以上例程的输出类似于:

```
string(9) "property1"
string(19) "Public property one"
```

```
string(9) "property2"
string(19) "Public property two"

string(9) "property3"
string(21) "Public property three"

string(9) "property4"
string(13) "last property"
```

IteratorAggregate::getIterator

IteratorAggregate::getIterator —该接口实现了 Iterator 或 Traversable 接口的类的一个实例，可以获取并返回一个外部迭代器，此函数没有参数，失败时抛出 Exception。

69.4 ArrayAccess interface

ArrayAccess interface 提供像访问数组一样访问对象的能力的接口。

```
ArrayAccess {
  /* 方法 */
  abstract public boolean offsetExists ( mixed $offset )
  abstract public mixed offsetGet ( mixed $offset )
  abstract public void offsetSet ( mixed $offset , mixed $value )
  abstract public void offsetUnset ( mixed $offset )
}
```

ArrayAccess **接口方法**

- ArrayAccess::offsetExists —检查一个偏移位置是否存在
- ArrayAccess::offsetGet —获取一个偏移位置的值
- ArrayAccess::offsetSet —设置一个偏移位置的值
- ArrayAccess::offsetUnset —复位一个偏移位置的值

Example 55 Basic usage

```
<?php
class obj implements arrayaccess {
   private $container = array();
   public function __construct() {
       $this->container = array(
           "one"  => 1,
           "two"  => 2,
           "three" => 3,
```

```
        );
    }
    public function offsetSet($offset, $value) {
        if (is_null($offset)) {
            $this->container[] = $value;
        } else {
            $this->container[$offset] = $value;
        }
    }
    public function offsetExists($offset) {
        return isset($this->container[$offset]);
    }
    public function offsetUnset($offset) {
        unset($this->container[$offset]);
    }
    public function offsetGet($offset) {
        return isset($this->container[$offset]) ? $this->container[$offset] :
            null;
    }
}

$obj = new obj;

var_dump(isset($obj["two"]));
var_dump($obj["two"]);
unset($obj["two"]);
var_dump(isset($obj["two"]));
$obj["two"] = "A value";
var_dump($obj["two"]);
$obj[] = 'Append 1';
$obj[] = 'Append 2';
$obj[] = 'Append 3';
print_r($obj);
?>
```

以上例程的输出类似于:

```
bool(true)
int(2)
bool(false)
string(7) "A value"
```

```
obj Object
(
    [container:obj:private] => Array
        (
            [one] => 1
            [three] => 3
            [two] => A value
            [0] => Append 1
            [1] => Append 2
            [2] => Append 3
        )

)
```

ArrayAccess::offsetExists

ArrayAccess::offsetExists 一检查一个偏移位置是否存在。

```
abstract public boolean ArrayAccess::offsetExists ( mixed $offset )
```

- offset 需要检查的偏移位置。

对一个实现了 ArrayAccess 接口的对象使用 isset() 或 empty() 时，此方法将执行，成功时返回 TRUE，或者在失败时返回 FALSE。如果一个非布尔型返回值被返回，将被转换为布尔型。

当使用 empty() 并且仅当 ArrayAccess::offsetExists() 返回 TRUE 时，ArrayAccess::offsetGet() 将被调用以检查是为否空。

Example 56 ArrayAccess::offsetExists() 范例

```
<?php
class obj implements arrayaccess {
   public function offsetSet($offset, $value) {
      var_dump(__METHOD__);
   }
   public function offsetExists($var) {
      var_dump(__METHOD__);
      if ($var == "foobar") {
         return true;
      }
      return false;
   }
```

```
    public function offsetUnset($var) {
        var_dump(__METHOD__);
    }
    public function offsetGet($var) {
        var_dump(__METHOD__);
        return "value";
    }
}

$obj = new obj;

echo "Runs obj::offsetExists()\n";
var_dump(isset($obj["foobar"]));

echo "\nRuns obj::offsetExists() and obj::offsetGet()\n";
var_dump(empty($obj["foobar"]));

echo "\nRuns obj::offsetExists(), *not* obj:offsetGet() as there is nothing
    to get\n";
var_dump(empty($obj["foobaz"]));
?>
```

以上例程的输出类似于:

```
Runs obj::offsetExists()
string(17) "obj::offsetExists"
bool(true)

Runs obj::offsetExists() and obj::offsetGet()
string(17) "obj::offsetExists"
string(14) "obj::offsetGet"
bool(false)

Runs obj::offsetExists(), *not* obj:offsetGet() as there is nothing to get
string(17) "obj::offsetExists"
bool(true)
```

ArrayAccess::offsetGet

ArrayAccess::offsetGet —获取并返回指定偏移位置的值，可返回任何类型。

```
abstract public mixed ArrayAccess::offsetGet ( mixed $offset )
```

- offset 需要获取的偏移位置。

当检查一个偏移位置是否为 empty() 时，此方法被执行。

Starting with PHP 5.3.4, the prototype checks were relaxed and it's possible for implementations of this method to return by reference. This makes indirect modifications to the overloaded array dimensions of ArrayAccess objects possible.

A direct modification is one that replaces completely the value of the array dimension, as in $obj[6] = 7. An indirect modification, on the other hand, only changes part of the dimension, or attempts to assign the dimension by reference to another variable, as in $obj[6][7] = 7 or $var =& $obj[6]. Increments with ++ and decrements with – are also implemented in a way that requires indirect modification.

While direct modification triggers a call to ArrayAccess::offsetSet(), indirect modification triggers a call to ArrayAccess::offsetGet(). In that case, the implementation of ArrayAccess::offsetGet() must be able to return by reference, otherwise an E_NOTICE message is raised.

ArrayAccess::offsetSet

ArrayAccess::offsetSet —设置一个偏移位置的值

```
abstract public void ArrayAccess::offsetSet ( mixed $offset , mixed $value )
```

为指定的偏移位置设置一个值，并且没有返回值。

- offset 待设置的偏移位置。
- value 需要设置的值。

如果另一个值不可用，那么 offset 参数将被设置为 NULL，就像下面的例子。

```
<?php
$arrayaccess[] = "first value";
$arrayaccess[] = "second value";
print_r($arrayaccess);
?>
```

以上例程会输出：

```
Array
(
    [0] => first value
    [1] => second value
)
```

ArrayAccess::offsetUnset

ArrayAccess::offsetUnset —复位一个偏移位置的值，执行时没有返回值。

```
abstract public void ArrayAccess::offsetUnset ( mixed $offset )
```

- offset 待复位的偏移位置。

  当使用 (unset) 进行类型转换时，该方法不会被调用。

69.5 Serializable interface

自定义序列化的接口。

实现此接口的类将不再支持 __sleep() 和 __wakeup()。不论何时，只要有实例需要被序列化，serialize 方法都将被调用。它将不会调用 __destruct() 或有其他影响，除非程序化地调用此方法。当数据被反序列化时，类将被感知并且调用合适的 unserialize() 方法而不是调用 __construct()。如果需要执行标准的构造器，你应该在这个方法中进行处理。

```
Serializable {
  /* 方法 */
  abstract public string serialize ( void )
  abstract public mixed unserialize ( string $serialized )
}
```

- Serializable::serialize —对象的字符串表示
- Serializable::unserialize —构造对象

Example 57 Basic usage

```
<?php
class obj implements Serializable {
    private $data;
    public function __construct() {
        $this->data = "My private data";
    }
    public function serialize() {
        return serialize($this->data);
    }
    public function unserialize($data) {
        $this->data = unserialize($data);
    }
    public function getData() {
        return $this->data;
    }
}

$obj = new obj;
```

```
$ser = serialize($obj);

$newobj = unserialize($ser);

var_dump($newobj->getData());
?>
```

以上例程的输出类似于:

```
string(15) "My private data"
```

Serializable::serialize

Serializable::serialize —返回对象的字符串表示或者 NULL，此函数没有参数。如果返回除了字符串或 NULL 之外的其他类型，将抛出 Exception。

```
abstract public string Serializable::serialize ( void )
```

这个方法担当着对象析构器的角色。在此方法之后，__destruct() 方法将不会被调用，返回没有序列化之前的原始值。

Serializable::unserialize

Serializable::unserialize —构造对象，在反序列化对象时被调用。

```
abstract public mixed Serializable::unserialize ( string $serialized )
```

- serialized 对象的字符串表示

这个方法担当着对象构造器的角色。在此方法之后，__construct() 将不会被调用。

Chapter 70

Predefined Class

70.1 Closure class

Closure 类用于代表匿名函数的类，匿名函数（在 PHP 5.3 中被引入）会产生这个类型的对象。

在过去，这个类被认为是一个实现细节，但现在可以依赖它做一些事情。自 PHP 5.4 起，这个类带有一些方法，允许在匿名函数创建后对其进行更多的控制。

除了此处列出的方法，还有一个 __invoke 方法。这是为了与其他实现了 __invoke() 魔术方法的对象保持一致性，但调用匿名函数的过程与它无关。

```
Closure {
  /* 方法 */
  __construct ( void )
  public static Closure bind ( Closure $closure , object $newthis [, mixed $newscope = '
      static' ] )
  public Closure bindTo ( object $newthis [, mixed $newscope = 'static' ] )
}
```

- Closure::__construct —用于禁止实例化的构造函数
- Closure::bind —复制一个闭包，绑定指定的 $this 对象和类作用域。
- Closure::bindTo —复制当前闭包对象，绑定指定的 $this 对象和类作用域。

Closure::__construct

Closure::__construct —用于禁止实例化的构造函数，此函数没有参数。

```
Closure::__construct ( void )
```

这个方法仅用于禁止实例化一个 Closure 类的对象。这个类的对象的创建方法写在匿名函数页。

该方法没有返回值，它只是简单的触发一个错误（类型是 E_RECOVERABLE_ERROR）。

Closure::bind

Closure::bind 一复制一个闭包，绑定指定的 $this 对象和类作用域。

```
public static Closure Closure::bind ( Closure $closure , object $newthis [, mixed
    $newscope = 'static' ] )
```

这个方法是 Closure::bindTo() 的静态版本，用于返回一个新的 Closure 对象或者在失败时返回 FALSE。

- closure 需要绑定的匿名函数。
- newthis 需要绑定到匿名函数的对象，或者 NULL 创建未绑定的闭包。
- newscope 想要绑定给闭包的类作用域，或者'static' 表示不改变。如果传入一个对象，则使用这个对象的类型名。类作用域用来决定在闭包中 $this 对象的私有、保护方法的可见性。

  The class scope to which associate the closure is to be associated, or 'static' to keep the current one. If an object is given, the type of the object will be used instead. This determines the visibility of protected and private methods of the bound object.

Example 58 Closure::bind() 实例

```
<?php
class A {
    private static $sfoo = 1;
    private $ifoo = 2;
}
$cl1 = static function() {
    return A::$sfoo;
};
$cl2 = function() {
    return $this->ifoo;
};

$bcl1 = Closure::bind($cl1, null, 'A');
$bcl2 = Closure::bind($cl2, new A(), 'A');
echo $bcl1(), "\n";
echo $bcl2(), "\n";
?>
```

以上例程的输出类似于：

```
1
2
```

Closure::bindTo

Closure::bindTo —复制当前闭包对象，绑定指定的 $this 对象和类作用域，返回新创建的 Closure 对象或者在失败时返回 FALSE。

```
public Closure Closure::bindTo ( object $newthis [, mixed $newscope = 'static' ] )
```

创建并返回一个匿名函数，它与当前对象的函数体相同、绑定了同样变量，但可以绑定不同的对象，也可以绑定新的类作用域。

如果你只是想要复制一个匿名函数，可以用 cloning 代替。

“绑定的对象”决定了函数体中的 $this 的取值，“类作用域”代表一个类型、决定在这个匿名函数中能够调用哪些私有和保护的方法。也就是说，此时 $this 可以调用的方法，与 newscope 类的成员函数是相同的。

静态闭包不能有绑定的对象 (newthis 参数的值应该设为 NULL) 不过仍然可以用 bubdTo 方法来改变它们的类作用域。

This function will ensure that for a non-static closure, having a bound instance will imply being scoped and vice-versa. To this end, non-static closures that are given a scope but a NULL instance are made static and non-static non-scoped closures that are given a non-null instance are scoped to an unspecified class.

- newthis
  绑定给匿名函数的一个对象，或者 NULL 来取消绑定。
- newscope
  关联到匿名函数的类作用域，或者’static’ 保持当前状态。如果是一个对象，则使用这个对象的类型为心得类作用域。这会决定绑定的对象的保护、私有成员方法的可见性。

Example 59 Closure::bindTo() 实例

```
<?php

class A {
    function __construct($val) {
        $this->val = $val;
    }
    function getClosure() {
        //returns closure bound to this object and scope
        return function() { return $this->val; };
    }
}

$ob1 = new A(1);
$ob2 = new A(2);
```

```
$cl = $ob1->getClosure();
echo $cl(), "\n";
$cl = $cl->bindTo($ob2);
echo $cl(), "\n";
?>
```

以上例程的输出类似于:

```
1
2
```

Chapter 71

Anonymous Class

匿名类可以用来创建一次性的简单对象。

```
<?php
// PHP 7 之前的代码
class Logger
{
    public function log($msg)
    {
        echo $msg;
    }
}

$util->setLogger(new Logger());

// 使用了 PHP 7+ 后的代码
$util->setLogger(new class {
    public function log($msg)
    {
        echo $msg;
    }
});
```

可以传递参数到匿名类的构造器，也可以扩展（extend）其他类、实现接口（implement interface），以及普通的类一样使用 trait。

```
<?php

class SomeClass {}
interface SomeInterface {}
```

```
trait SomeTrait {}

var_dump(new class(10) extends SomeClass implements SomeInterface {
    private $num;

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

    use SomeTrait;
});
```

以上示例会输出：

```
object(class@anonymous)#1 (1) {
  ["Command line code0x104c5b612":"class@anonymous":private]=>
  int(10)
}
```

匿名类被嵌套进普通 Class 后，不能访问这个外部类（Outer class）的 private（私有）、protected（受保护）方法或者属性。

- 为了访问外部类（Outer class）protected 属性或方法，匿名类可以 extend（扩展）外部类。
- 为了使用外部类（Outer class）的 private 属性，必须通过构造器传入匿名类。

```
<?php
class Outer
{
    private $prop = 1;
    protected $prop2 = 2;

    protected function func1()
    {
        return 3;
    }

    public function func2()
    {
        return new class($this->prop) extends Outer {
            private $prop3;

            public function __construct($prop)
```

```
            {
                $this->prop3 = $prop;
            }

            public function func3()
            {
                return $this->prop2 + $this->prop3 + $this->func1();
            }
        };
    }
}

echo (new Outer)->func2()->func3();
```

以上示例会输出：

```
6
```

Chapter 72

Traits

PHP 使用 trait 实现代码复用。

Trait 是为单继承语言而准备的一种代码复用机制，可以减少单继承语言的限制来自由地在不同层次结构内的独立的类中复用方法。

Trait 和 Class 组合的语义都是定义了一种方式来减少复杂性，避免传统多继承和 Mixin 类相关的典型问题。

除了和 Class 的相似之处之外，Trait 仅仅旨在用细粒度和一致的方式来组合功能。

Trait 不能通过它自身来实例化，从而为传统继承增加了水平特性的组合。或者说，应用 Trait 的 Class 之间不需要继承，直接注入使用。

```
<?php
trait ezcReflectionReturnInfo {
    function getReturnType() { /*1*/ }
    function getReturnDescription() { /*2*/ }
}

class ezcReflectionMethod extends ReflectionMethod {
    use ezcReflectionReturnInfo;
    /* ... */
}

class ezcReflectionFunction extends ReflectionFunction {
    use ezcReflectionReturnInfo;
    /* ... */
}
?>
```

72.1 Precedence

从基类继承的成员将被 trait 注入的成员所覆盖。优先顺序是来自当前类的成员覆盖了 trait 的方法，trait 则覆盖了被继承的方法。

例如，从基类继承的成员被注入的 SayWorld Trait 中的 MyHelloWorld 方法所覆盖，其行为和 MyHelloWorld 类中定义的方法一致。优先顺序是当前类中的方法会覆盖 trait 方法，而 trait 方法则覆盖基类中的方法。

```
<?php
class Base {
   public function sayHello() {
      echo 'Hello ';
   }
}

trait SayWorld {
   public function sayHello() {
      parent::sayHello();
      echo 'World!';
   }
}

class MyHelloWorld extends Base {
   use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
?>
```

以上例程会输出：

```
Hello World!
```

```
<?php
trait HelloWorld {
   public function sayHello() {
      echo 'Hello World!';
   }
}

class TheWorldIsNotEnough {
```

```
    use HelloWorld;
    public function sayHello() {
        echo 'Hello Universe!';
    }
}

$o = new TheWorldIsNotEnough();
$o->sayHello();
?>
```

以上例程会输出：

```
Hello Universe!
```

72.2 Multiple Traits

在 use 声明列出多个 trait，通过逗号分隔就可以都插入到一个类中。

```
<?php
trait Hello {
    public function sayHello() {
        echo 'Hello ';
    }
}

trait World {
    public function sayWorld() {
        echo 'World';
    }
}

class MyHelloWorld {
    use Hello, World;
    public function sayExclamationMark() {
        echo '!';
    }
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

```
?>
```

以上例程会输出：

```
Hello World!
```

72.3 Conflict Resolution

如果两个 trait 都插入了一个同名的方法，如果没有明确解决冲突将会产生一个致命错误。

为了解决多个 trait 在同一个类中的命名冲突，需要使用 insteadof 操作符来明确指定使用冲突方法中的哪一个。

上述方式仅允许排除掉其它方法，as 操作符则可以将其中一个冲突的方法以另一个名称来引入。

在本例中 Talker 使用了 trait A 和 B，A 和 B 有冲突的方法被指定使用 trait B 中的 smallTalk 以及 trait A 中的 bigTalk。

Aliased_Talker 使用了 as 操作符来定义了 talk 来作为 B 的 bigTalk 的别名。

```
<?php
trait A {
    public function smallTalk() {
        echo 'a';
    }
    public function bigTalk() {
        echo 'A';
    }
}

trait B {
    public function smallTalk() {
        echo 'b';
    }
    public function bigTalk() {
        echo 'B';
    }
}

class Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
```

```
    }
}

class Aliased_Talker {
    use A, B {
        B::smallTalk insteadof A;
        A::bigTalk insteadof B;
        B::bigTalk as talk;
    }
}
?>
```

72.4 Method Visibility

as 语法还可以用来调整方法的访问控制。

```
<?php
trait HelloWorld {
    public function sayHello() {
        echo 'Hello World!';
    }
}

// 修改 sayHello 的访问控制
class MyClass1 {
    use HelloWorld {
      sayHello as protected;
    }
}

// 给方法一个改变了访问控制的别名
// 原版 sayHello 的访问控制则没有发生变化
class MyClass2 {
    use HelloWorld {
      sayHello as private myPrivateHello;
    }
}
?>
```

72.5 Traits Compose

除了在 Class 中可以使用 Trait，其它 trait 也能够使用 trait。

在 trait 定义时通过使用一个或多个 trait 能够组合其它 trait 中的部分或全部成员。

```
<?php
trait Hello {
   public function sayHello() {
      echo 'Hello ';
   }
}

trait World {
   public function sayWorld() {
      echo 'World!';
   }
}

trait HelloWorld {
   use Hello, World;
}

class MyHelloWorld {
   use HelloWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
?>
```

以上例程会输出：

```
Hello World!
```

72.6 Abstract Trait Members

为了对使用的类施加强制要求，trait 支持抽象方法的使用。

```
通过抽象方法来进行强制要求
<?php
trait Hello {
```

```
    public function sayHelloWorld() {
        echo 'Hello'.$this->getWorld();
    }
    abstract public function getWorld();
}

class MyHelloWorld {
    private $world;
    use Hello;
    public function getWorld() {
        return $this->world;
    }
    public function setWorld($val) {
        $this->world = $val;
    }
}
?>
```

72.7 Static Trait Members

静态变量可以被 trait 的方法引用，但不能被 trait 定义。但是 trait 能够为使用的类定义静态方法。

Example 60 静态变量

```
<?php
trait Counter {
    public function inc() {
        static $c = 0;
        $c = $c + 1;
        echo "$c\n";
    }
}

class C1 {
    use Counter;
}

class C2 {
    use Counter;
}
```

```
$o = new C1(); $o->inc(); // echo 1
$p = new C2(); $p->inc(); // echo 1
?>
```

Example 61 静态方法

```
<?php
trait StaticExample {
    public static function doSomething() {
        return 'Doing something';
    }
}

class Example {
    use StaticExample;
}

Example::doSomething();
?>
```

72.8 Trait Properties

Trait 同样可以定义属性。

```
定义属性
<?php
trait PropertiesTrait {
    public $x = 1;
}

class PropertiesExample {
    use PropertiesTrait;
}

$example = new PropertiesExample;
$example->x;
?>
```

- 如果 trait 定义了一个属性，那么类将不能定义同样名称的属性，否则会产生一个错误。
- 如果该属性在类中的定义与在 trait 中的定义兼容（同样的可见性和初始值）则错误的级别是 E_STRICT，否则是一个致命错误。

Example 62 解决冲突

```
<?php
trait PropertiesTrait {
    public $same = true;
    public $different = false;
}

class PropertiesExample {
    use PropertiesTrait;
    public $same = true; // Strict Standards
    public $different = true; // 致命错误
}
?>
```

Chapter 73

Overloading

PHP 支持的” 重载”（overloading）是通过魔术方法实现动态地” 创建” 类属性和方法，而且魔术方法的参数不能通过引用传递。

PHP 的” 重载” 与其它绝大多数面向对象语言不同，传统的” 重载” 是用于提供多个同名的类方法，而且各个方法的参数类型和个数不同。

PHP 中的” 不可访问属性（inaccessible properties）” 和” 不可访问方法（inaccessible methods）” 是未定义或不可见的类属性或方法。当调用当前环境下未定义或不可见的类属性或方法时，重载方法会被调用。

所有的重载方法都必须被声明为 public，可见性未设置为 public 或未声明为 static 的时候可能会产生一个警告。

73.1 Property Overloading

属性重载只能在对象中进行。

```
public void __set ( string $name , mixed $value )
public mixed __get ( string $name )
public bool __isset ( string $name )
public void __unset ( string $name )
```

- 向不可访问属性赋值时，__set() 会被调用。
- 读取不可访问属性的值时，__get() 会被调用。
- 对不可访问属性调用 isset() 或 empty() 时，__isset() 会被调用。
- 对不可访问属性调用 unset() 时，__unset() 会被调用。

其中，参数 $name 是指要操作的变量名称，__set() 方法的 $value 参数指定了 $name 变量的值。

在静态方法中，上述这些魔术方法将不会被调用，因此这些魔术方法都不能被声明为 static，而且将这些魔术方法定义为 static 时会产生一个警告。

根据 PHP 处理赋值运算的方式，__set() 的返回值将被忽略。例如，在下面这样的链式赋值中，__get() 不会被调用。

```
$a = $obj->b = 8;
```

在除 isset() 之外的其它语言结构中无法使用重载的属性，因此对一个重载的属性使用 empty() 时，重载魔术方法将不会被调用。为了避开此限制，必须将重载属性赋值到本地变量后再使用 empty()。

Example 63 使用 __get()，__set()，__isset() 和 __unset() 进行属性重载

```
<?php
class PropertyTest {
    /** 被重载的数据保存在此 */
    private $data = array();


    /** 重载不能被用在已经定义的属性 */
    public $declared = 1;

    /** 只有从类外部访问这个属性时，重载才会发生 */
    private $hidden = 2;

    public function __set($name, $value)
    {
        echo "Setting '$name' to '$value'\n";
        $this->data[$name] = $value;
    }

    public function __get($name)
    {
        echo "Getting '$name'\n";
        if (array_key_exists($name, $this->data)) {
            return $this->data[$name];
        }

        $trace = debug_backtrace();
        trigger_error(
            'Undefined property via __get(): ' . $name .
            ' in ' . $trace[0]['file'] .
```

```
            ' on line ' . $trace[0]['line'],
            E_USER_NOTICE);
        return null;
    }

    /** PHP 5.1.0之后版本 */
    public function __isset($name)
    {
        echo "Is '$name' set?\n";
        return isset($this->data[$name]);
    }

    /** PHP 5.1.0之后版本 */
    public function __unset($name)
    {
        echo "Unsetting '$name'\n";
        unset($this->data[$name]);
    }

    /** 非魔术方法 */
    public function getHidden()
    {
        return $this->hidden;
    }
}


echo "<pre>\n";

$obj = new PropertyTest;

$obj->a = 1;
echo $obj->a . "\n\n";

var_dump(isset($obj->a));
unset($obj->a);
var_dump(isset($obj->a));
echo "\n";

echo $obj->declared . "\n\n";
```

```
echo "Let's experiment with the private property named 'hidden':\n";
echo "Privates are visible inside the class, so __get() not used...\n";
echo $obj->getHidden() . "\n";
echo "Privates not visible outside of class, so __get() is used...\n";
echo $obj->hidden . "\n";
?>
```

以上例程会输出：

```
Setting 'a' to '1'
Getting 'a'
1

Is 'a' set?
bool(true)
Unsetting 'a'
Is 'a' set?
bool(false)

1

Let's experiment with the private property named 'hidden':
Privates are visible inside the class, so __get() not used...
2
Privates not visible outside of class, so __get() is used...
Getting 'hidden'


Notice:  Undefined property via __get(): hidden in <file> on line 70 in <file> on line 29
```

73.2 Method Overloading

```
public mixed __call ( string $name , array $arguments )
public static mixed __callStatic ( string $name , array $arguments )
```

- 在对象中调用一个不可访问方法时，__call() 会被调用。
- 在静态上下文中调用一个不可访问方法时，__callStatic() 会被调用。

__callStatic() 魔术方法的可见性未设置为 public 或未声明为 static 的时候会产生一个警告。

其中，$name 参数是要调用的方法名称，$arguments 参数是一个枚举数组，包含着要传递给方法 $name 的参数。

Example 64　使用 __call() 和 __callStatic() 对方法重载

```
<?php
class MethodTest
{
    public function __call($name, $arguments)
    {
        // 注意: $name 的值区分大小写
        echo "Calling object method '$name' "
             . implode(', ', $arguments). "\n";
    }

    /** PHP 5.3.0之后版本 */
    public static function __callStatic($name, $arguments)
    {
        // 注意: $name 的值区分大小写
        echo "Calling static method '$name' "
             . implode(', ', $arguments). "\n";
    }
}

$obj = new MethodTest;
$obj->runTest('in object context');

MethodTest::runTest('in static context'); // PHP 5.3.0之后版本
?>
```

以上例程会输出：

```
Calling object method 'runTest' in object context
Calling static method 'runTest' in static context
```

Chapter 74

Object Iteration

74.1 Iterations

PHP 提供了一种定义对象的方法使其可以通过单元列表来遍历（例如用 foreach 语句）。默认情况下，所有可见属性都将被用于遍历。

Example 65 foreach 遍历对象的所有能够访问的可见属性

```
<?php
class MyClass
{
    public $var1 = 'value 1';
    public $var2 = 'value 2';
    public $var3 = 'value 3';

    protected $protected = 'protected var';
    private  $private  = 'private var';

    function iterateVisible() {
       echo "MyClass::iterateVisible:\n";
       foreach($this as $key => $value) {
           print "$key => $value\n";
       }
    }
}

$class = new MyClass();

foreach($class as $key => $value) {
```

```
    print "$key => $value\n";
}
echo "\n";


$class->iterateVisible();
?>
```

以上例程会输出：

```
var1 => value 1
var2 => value 2
var3 => value 3

MyClass::iterateVisible:
var1 => value 1
var2 => value 2
var3 => value 3
protected => protected var
private => private var
```

更进一步，可以实现 Iterator 接口来让对象自行决定如何遍历以及每次遍历时那些值可用。

Example 66 实现 Iterator 接口的对象遍历

```
<?php
class MyIterator implements Iterator
{
    private $var = array();

    public function __construct($array)
    {
        if (is_array($array)) {
            $this->var = $array;
        }
    }

    public function rewind() {
        echo "rewinding\n";
```

```
        reset($this->var);
    }

    public function current() {
        $var = current($this->var);
        echo "current: $var\n";
        return $var;
    }

    public function key() {
        $var = key($this->var);
        echo "key: $var\n";
        return $var;
    }

    public function next() {
        $var = next($this->var);
        echo "next: $var\n";
        return $var;
    }

    public function valid() {
        $var = $this->current() !== false;
        echo "valid: {$var}\n";
        return $var;
    }
}

$values = array(1,2,3);
$it = new MyIterator($values);

foreach ($it as $a => $b) {
    print "$a: $b\n";
}
?>
```

以上例程会输出:

```
rewinding
current: 1
valid: 1
```

```
current: 1
key: 0
0: 1
next: 2
current: 2
valid: 1
current: 2
key: 1
1: 2
next: 3
current: 3
valid: 1
current: 3
key: 2
2: 3
next:
current:
valid:
```

IteratorAggregate 接口可以替代实现所有的 Iterator 方法，IteratorAggregate 只需要实现一个方法 IteratorAggregate::getIterator() 就可以返回一个实现了 Iterator 的类的实例。

Example 67 实现 IteratorAggregate 来遍历对象

```php
<?php
class MyCollection implements IteratorAggregate
{
    private $items = array();
    private $count = 0;

    // Required definition of interface IteratorAggregate
    public function getIterator() {
        return new MyIterator($this->items);
    }

    public function add($value) {
        $this->items[$this->count++] = $value;
    }
}
```

```php
$coll = new MyCollection();
$coll->add('value 1');
$coll->add('value 2');
$coll->add('value 3');

foreach ($coll as $key => $val) {
    echo "key/value: [$key -> $val]\n\n";
}
?>
```

以上例程会输出：

```
rewinding
current: value 1
valid: 1
current: value 1
key: 0
key/value: [0 -> value 1]

next: value 2
current: value 2
valid: 1
current: value 2
key: 1
key/value: [1 -> value 2]

next: value 3
current: value 3
valid: 1
current: value 3
key: 2
key/value: [2 -> value 3]

next:
current:
valid:
```

另外，还可以通过生成器提供的方法来定义 Iterators。

74.2 Generators

生成器提供了一种更容易的方法来实现简单的对象迭代，同时没有实现一个具有 Iterator 接口的类所带来的性能开销和复杂性。

生成器允许在 foreach 代码块中写代码来迭代一组数据而不需要在内存中创建一个数组，否则会使内存达到上限，或者会占据可观的处理时间。

相反，用户可以写一个生成器函数，就像一个普通的自定义函数一样，不过和普通函数只返回一次不同的是生成器可以根据需要 yield 多次来生成需要迭代的值。

一个简单的例子就是使用生成器来重新实现 range() 函数。标准的 range() 函数需要为其中的每一个返回值在内存中生成一个数组，结果就是生成一个很大的数组。例如，调用 range(0, 1000000) 将导致内存占用超过 100 MB。

做为一种替代方法，我们可以实现一个 xrange() 生成器，只需要足够的内存来创建 Iterator 对象并在内部跟踪生成器的当前状态，这样只需要不到 1K 字节的内存。

Example 68 将 range() 实现为生成器

```
<?php
function xrange($start, $limit, $step = 1) {
    if ($start < $limit) {
        if ($step <= 0) {
            throw new LogicException('Step must be +ve');
        }

        for ($i = $start; $i <= $limit; $i += $step) {
            yield $i;
        }
    } else {
        if ($step >= 0) {
            throw new LogicException('Step must be -ve');
        }

        for ($i = $start; $i >= $limit; $i += $step) {
            yield $i;
        }
    }
}

/* Note that both range() and xrange() result in the same
 * output below. */

echo 'Single digit odd numbers from range(): ';
```

```
foreach (range(1, 9, 2) as $number) {
    echo "$number ";
}
echo "\n";

echo 'Single digit odd numbers from xrange(): ';
foreach (xrange(1, 9, 2) as $number) {
    echo "$number ";
}
?>
```

以上例程会输出：

```
Single digit odd numbers from range():  1 3 5 7 9
Single digit odd numbers from xrange(): 1 3 5 7 9
```

74.3 Generator syntax

A generator function looks just like a normal function, except that instead of returning a value, a generator yields as many values as it needs to.

When a generator function is called, it returns an object that can be iterated over. When you iterate over that object (for instance, via a foreach loop), PHP will call the generator function each time it needs a value, then saves the state of the generator when the generator yields a value so that it can be resumed when the next value is required.

Once there are no more values to be yielded, then the generator function can simply exit, and the calling code continues just as if an array has run out of values.

A generator cannot return a value: doing so will result in a compile error. An empty return statement is valid syntax within a generator and it will terminate the generator.

74.4 yield keyword

The heart of a generator function is the yield keyword. In its simplest form, a yield statement looks much like a return statement, except that instead of stopping execution of the function and returning, yield instead provides a value to the code looping over the generator and pauses execution of the generator function.

Example 69 A simple example of yielding values

```php
<?php
function gen_one_to_three() {
    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $i;
    }
}

$generator = gen_one_to_three();
foreach ($generator as $value) {
    echo "$value\n";
}
?>
```

以上例程会输出：

```
1
2
3
```

Internally, sequential integer keys will be paired with the yielded values, just as with a non-associative array.

If you use yield in an expression context (for example, on the right hand side of an assignment), you must surround the yield statement with parentheses. For example, this is valid:

```php
$data = (yield $value);
```

But this is not, and will result in a parse error:

```php
$data = yield $value;
```

This syntax may be used in conjunction with the send() method on Generator objects.

74.5 Yielding values with keys

PHP also supports associative arrays, and generators are no different. In addition to yielding simple values, as shown above, you can also yield a key at the same time.

The syntax for yielding a key/value pair is very similar to that used to define an associative array, as shown below.

Example 70 Yielding a key/value pair

```
<?php
/* The input is semi-colon separated fields, with the first
 * field being an ID to use as a key. */

$input = <<<'EOF'
1;PHP;Likes dollar signs
2;Python;Likes whitespace
3;Ruby;Likes blocks
EOF;

function input_parser($input) {
    foreach (explode("\n", $input) as $line) {
        $fields = explode(';', $line);
        $id = array_shift($fields);

        yield $id => $fields;
    }
}

foreach (input_parser($input) as $id => $fields) {
    echo "$id:\n";
    echo "   $fields[0]\n";
    echo "   $fields[1]\n";
}
?>
```

以上例程会输出：

```
1:
    PHP
    Likes dollar signs
2:
    Python
    Likes whitespace
3:
    Ruby
    Likes blocks
```

As with the simple value yields shown earlier, yielding a key/value pair in an expression context requires the yield statement to be parenthesised:

```
$data = (yield $key => $value);
```

74.6 Yielding null values

Yield can be called without an argument to yield a NULL value with an automatic key.

Example 71 Yielding NULLs

```
<?php
function gen_three_nulls() {
    foreach (range(1, 3) as $i) {
        yield;
    }
}

var_dump(iterator_to_array(gen_three_nulls()));
?>
```

以上例程会输出：

```
array(3) {
  [0]=>
  NULL
  [1]=>
  NULL
  [2]=>
  NULL
}
```

74.7 Yielding by reference

Generator functions are able to yield values by reference as well as by value. This is done in the same way as returning references from functions: by prepending an ampersand to the function name.

Example 72 Yielding values by reference

```
<?php
function &gen_reference() {
    $value = 3;
```

```
    while ($value > 0) {
        yield $value;
    }
}

/* Note that we can change $number within the loop, and
 * because the generator is yielding references, $value
 * within gen_reference() changes. */
foreach (gen_reference() as &$number) {
    echo (--$number).'... ';
}
?>
```

以上例程会输出：

```
2... 1... 0...
```

74.8 Generator objects

When a generator function is called for the first time, an object of the internal Generator class is returned. This object implements the Iterator interface in much the same way as a forward-only iterator object would.

Most methods in the Generator class have the same semantics as the methods in the Iterator interface, but generator objects also have one additional method: send().

Generator objects cannot be instantiated via new.

Example 73 The Generator class

```
<?php
class Generator implements Iterator {
    public function rewind();     // Rewinds the iterator. If
                                  // iteration has already begun,
                                  // this will throw an exception.

    public function valid();      // Returns false if the
                                  // iterator has been closed.
                                  // Otherwise returns true.

    public function current();    // Returns the yielded value.
```

```
    public function key();        // Returns the yielded key.

    public function next();       // Resumes execution of the
                                  // generator.

    public function send($value); // Sends the given value to the
                                  // generator as the result of
                                  // the yield expression and
                                  // resumes execution of the
                                  // generator.
}
?>
```

74.9 Generator::send()

Generator::send() allows values to be injected into generator functions while iterating over them. The injected value will be returned from the yield statement and can then be used like any other variable within the generator function.

Example 74 Using Generator::send() to inject values

```
<?php
function printer() {
    while (true) {
        $string = yield;
        echo $string;
    }
}

$printer = printer();
$printer->send('Hello world!');
?>
```

以上例程会输出：

```
Hello world!
```

74.10 Generators & Iterator objects

The primary advantage of generators is their simplicity. Much less boilerplate code has to be written compared to implementing an Iterator class, and the code is generally much more readable. For example, the following function and class are equivalent:

Example 75 Comparing generators with Iterator objects

```
<?php
function getLinesFromFile($fileName) {
    if (!$fileHandle = fopen($fileName, 'r')) {
        return;
    }

    while (false !== $line = fgets($fileHandle)) {
        yield $line;
    }

    fclose($fileHandle);
}

// versus...

class LineIterator implements Iterator {
    protected $fileHandle;

    protected $line;
    protected $i;

    public function __construct($fileName) {
        if (!$this->fileHandle = fopen($fileName, 'r')) {
            throw new RuntimeException('Couldn\'t open file "' . $fileName . '
                "');
        }
    }

    public function rewind() {
        fseek($this->fileHandle, 0);
        $this->line = fgets($this->fileHandle);
        $this->i = 0;
    }
```

```
    public function valid() {
        return false !== $this->line;
    }

    public function current() {
        return $this->line;
    }

    public function key() {
        return $this->i;
    }

    public function next() {
        if (false !== $this->line) {
            $this->line = fgets($this->fileHandle);
            $this->i++;
        }
    }

    public function __destruct() {
        fclose($this->fileHandle);
    }
}
?>
```

This flexibility does come at a cost, however: generators are forward-only iterators, and cannot be rewound once iteration has started. This also means that the same generator can't be iterated over multiple times: the generator will need to either be rebuilt by calling the generator function again, or cloned via the `clone` keyword.

Example 76

Chapter 75

Magic Methods

PHP 的“魔术方法”(Magic methods)包括 __construct()、__destruct()、__call()、__callStatic()、__get()、__set()、__isset()、__unset()、__sleep()、__wakeup()、__toString()、__invoke()、__set_state() 、 __clone() 和 __debugInfo() 等。

所有以 __（两个下划线）开头的类方法都被保留为魔术方法，实践中除了上述魔术方法之外建议不要以 __ 为前缀。

在命名自己的类方法时不能使用魔术方法名，除非是需要使用其魔术功能。

75.1 __sleep()

```
public array __sleep ( void )
void __wakeup ( void )
```

serialize() 函数会检查类中是否存在一个魔术方法 __sleep()。如果存在，该方法会先被调用，然后才执行序列化操作。

上述功能可以用于清理对象，并返回一个包含对象中所有应被序列化的变量名称的数组。如果该方法未返回任何内容，则 NULL 被序列化，并产生一个 E_NOTICE 级别的错误。

__sleep() 不能返回父类的私有成员的名字，否则产生一个 E_NOTICE 级别的错误，可以用 Serializable 接口来替代。

__sleep() 方法通常用于提交未提交的数据或类似的清理操作，同时如果有一些很大的对象，但是不需要全部保存，这个功能可以用来进行清理。

75.2 __wakeup()

与 __sleep() 相反，unserialize() 会检查是否存在一个 __wakeup() 方法。如果存在，则会先调用 __wakeup 方法，预先准备对象需要的资源。

__wakeup() 经常用在反序列化操作中，例如重新建立数据库连接，或执行其它初始化操作。

```
<?php
class Connection
{
    protected $link;
    private $server, $username, $password, $db;

    public function __construct($server, $username, $password, $db)
    {
        $this->server = $server;
        $this->username = $username;
        $this->password = $password;
        $this->db = $db;
        $this->connect();
    }

    private function connect()
    {
        $this->link = mysql_connect($this->server, $this->username, $this->password);
        mysql_select_db($this->db, $this->link);
    }

    public function __sleep()
    {
        return array('server', 'username', 'password', 'db');
    }

    public function __wakeup()
    {
        $this->connect();
    }
}
?>
```

75.3 __toString()

```
public string __toString ( void )
```

__toString() 方法用于一个类被当成字符串时应怎样回应。例如，`echo $obj;` 应该显示些什么。

__toString() 方法必须返回一个字符串，否则将产生一条 E_RECOVERABLE_ERROR 级别的致命错误。

不能在 __toString() 方法中抛出异常，否则会导致致命错误。

```
<?php
// Declare a simple class
class TestClass
{
    public $foo;

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

    public function __toString() {
        return $this->foo;
    }
}

$class = new TestClass('Hello');
echo $class;
?>
```

以上例程会输出：

```
Hello
```

- 在 PHP 5.2.0 之前，__toString() 方法只有在直接使用于 echo 或 print 时才能生效。
- 在 PHP 5.2.0 之后，则可以在任何字符串环境生效（例如通过 printf()，使用%s 修饰符），但是不能用于非字符串环境（例如使用%d 修饰符）。

如果将一个未定义 __toString() 方法的对象转换为字符串，则会产生 E_RECOVERABLE_ERROR 级别的错误。

75.4 __invoke()

```
mixed __invoke ([ $... ] )
```

当尝试以调用函数的方式调用一个对象时，__invoke() 方法会被自动调用。

```php
<?php
class CallableClass
{
    function __invoke($x) {
        var_dump($x);
    }
}
$obj = new CallableClass;
$obj(5);
var_dump(is_callable($obj));
?>
```

以上例程会输出：

```
int(5)
bool(true)
```

75.5 __set_state()

```php
static object __set_state ( array $properties )
```

当调用 var_export() 导出类时，__set_state() 就会被调用。

__set_state() 方法的唯一参数是一个数组，其中包含按 `array('property' => value, ...)` 格式排列的类属性。

```php
<?php
class A
{
    public $var1;
    public $var2;

    public static function __set_state($an_array) // As of PHP 5.1.0
    {
        $obj = new A;
        $obj->var1 = $an_array['var1'];
        $obj->var2 = $an_array['var2'];
        return $obj;
    }
}

$a = new A;
$a->var1 = 5;
```

```
$a->var2 = 'foo';

eval('$b = ' . var_export($a, true) . ';'); // $b = A::__set_state(array(
                                            //   'var1' => 5,
                                            //   'var2' => 'foo',
                                            // ));
var_dump($b);
?>
```

以上例程会输出:

```
object(A)#2 (2) {
  ["var1"]=>
  int(5)
  ["var2"]=>
  string(3) "foo"
}
```

75.6 __debugInfo()

```
array __debugInfo ( void )
```

在使用 var_dump() 导出对象信息时，就会调用 __debugInfo() 方法。

如果对象内部没有定义 __debugInfo() 方法，那么所有的 public、protected 和 private 属性都会被输出。

```
<?php
class C {
   private $prop;

   public function __construct($val) {
      $this->prop = $val;
   }

   public function __debugInfo() {
      return [
         'propSquared' => $this->prop ** 2,
      ];
   }
}
```

```
var_dump(new C(42));
?>
```

Chapter 76

Final Keyword

- 如果父类中的一个方法被声明为 final，则子类无法覆盖该方法。
- 如果一个类被声明为 final，则这个不能被继承。

```
<?php
class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }

   final public function moreTesting() {
       echo "BaseClass::moreTesting() called\n";
   }
}

class ChildClass extends BaseClass {
   public function moreTesting() {
       echo "ChildClass::moreTesting() called\n";
   }
}
// Results in Fatal error: Cannot override final method BaseClass::moreTesting()
?>
```

属性不能被定义为 final，只有类和方法才能被定义为 final。

```
<?php
final class BaseClass {
   public function test() {
       echo "BaseClass::test() called\n";
   }
```

```
    // 这里无论你是否将方法声明为final，都没有关系
    final public function moreTesting() {
        echo "BaseClass::moreTesting() called\n";
    }
}

class ChildClass extends BaseClass {
}
// 产生 Fatal error: Class ChildClass may not inherit from final class (BaseClass)
?>
```

Chapter 77

Object Cloning

在多数情况下，实际上并不需要完全复制一个对象来获得其中属性，但是有一个情况下确实需要：如果有一个 GTK 窗口对象，该对象持有窗口相关的资源，如果需要复制一个新的窗口并保持所有属性与原来的窗口相同，但是必须是一个新的对象（因为如果不是新的对象，那么一个窗口中的改变就会影响到另一个窗口）。

还有一种情况就是，如果对象 A 中保存着对象 B 的引用，当复制对象 A 时需要让其中使用的对象不再是对象 B 而是 B 的一个副本，那么就必须得到对象 A 的一个副本。

对象复制可以通过 clone 关键字来完成（如果可能就会调用对象的 __clone() 方法），而且对象中的 __clone() 方法不能被直接调用。

```
$copy_of_object = clone $object;
```

当对象被复制后，PHP 会对对象的所有属性执行一个浅复制（shallow copy），所有的引用属性仍然会是一个指向原来的变量的引用。

```
void __clone ( void )
```

当对象复制完成时，如果定义了 __clone() 方法，则新创建的对象（复制生成的对象）中的 __clone() 方法会被调用，可以用来修改属性的值（如果有必要的话）。

```
<?php
class SubObject
{
   static $instances = 0;
   public $instance;

   public function __construct() {
      $this->instance = ++self::$instances;
   }
```

```
    public function __clone() {
        $this->instance = ++self::$instances;
    }
}

class MyCloneable
{
    public $object1;
    public $object2;

    function __clone()
    {

        // 强制复制一份this->object,  否则仍然指向同一个对象
        $this->object1 = clone $this->object1;
    }
}

$obj = new MyCloneable();

$obj->object1 = new SubObject();
$obj->object2 = new SubObject();

$obj2 = clone $obj;


print("Original Object:\n");
print_r($obj);

print("Cloned Object:\n");
print_r($obj2);

?>
```

以上例程会输出：

```
Original Object:
MyCloneable Object
(
    [object1] => SubObject Object
        (
```

```
            [instance] => 1
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )

)
Cloned Object:
MyCloneable Object
(
    [object1] => SubObject Object
        (
            [instance] => 3
        )

    [object2] => SubObject Object
        (
            [instance] => 2
        )

)
```

Chapter 78

Objects Comparison

PHP 5 中的对象比较要比 PHP 4 中复杂，而且所能预期的结果更符合一个面向对象语言。

- 如果使用比较运算符（==）比较两个对象变量，比较的原则是：如果两个对象的属性和属性值都相等，而且两个对象是同一个类的实例，那么这两个对象变量相等。
- 如果使用全等运算符（===），那么这两个对象变量一定要指向某个类的同一个实例（即同一个对象）。

```
<?php
function bool2str($bool)
{
    if ($bool === false) {
        return 'FALSE';
    } else {
        return 'TRUE';
    }
}

function compareObjects(&$o1, &$o2)
{
    echo 'o1 == o2 : ' . bool2str($o1 == $o2) . "\n";
    echo 'o1 != o2 : ' . bool2str($o1 != $o2) . "\n";
    echo 'o1 === o2 : ' . bool2str($o1 === $o2) . "\n";
    echo 'o1 !== o2 : ' . bool2str($o1 !== $o2) . "\n";
}

class Flag
{
```

```
    public $flag;

    function Flag($flag = true) {
        $this->flag = $flag;
    }
}

class OtherFlag
{
    public $flag;

    function OtherFlag($flag = true) {
        $this->flag = $flag;
    }
}

$o = new Flag();
$p = new Flag();
$q = $o;
$r = new OtherFlag();

echo "Two instances of the same class\n";
compareObjects($o, $p);

echo "\nTwo references to the same instance\n";
compareObjects($o, $q);

echo "\nInstances of two different classes\n";
compareObjects($o, $r);
?>
```

以上例程会输出：

```
Two instances of the same class
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : FALSE
o1 !== o2 : TRUE

Two references to the same instance
o1 == o2 : TRUE
```

```
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE

Instances of two different classes
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE
```

PHP 扩展中可以自行定义对象比较的原则。

Chapter 79

Type Hinting

PHP 可以使用类型约束。例如，函数的参数可以指定必须为对象（在函数原型里面指定类的名字），接口、数组或者 callable。不过，如果使用 NULL 作为参数的默认值，那么在调用函数的时候依然可以使用 NULL 作为实参。

如果一个类或接口指定了类型约束，则其所有的子类或实现也都必须如此。

类型约束不能用于标量类型（例如 int 或 string），Traits 也不允许。

```
<?php
class MyClass
{
    /**
     * 测试函数
     * 第一个参数必须为 OtherClass 类的一个对象
     */
    public function test(OtherClass $otherclass) {
        echo $otherclass->var;
    }


    /**
     * 另一个测试函数
     * 第一个参数必须为数组
     */
    public function test_array(array $input_array) {
        print_r($input_array);
    }
}
```

```
    /**
     * 第一个参数必须为递归类型
     */
    public function test_interface(Traversable $iterator) {
        echo get_class($iterator);
    }

    /**
     * 第一个参数必须为回调类型
     */
    public function test_callable(callable $callback, $data) {
        call_user_func($callback, $data);
    }
}

// OtherClass 类定义
class OtherClass {
    public $var = 'Hello World';
}
?>
```

如果函数调用的参数与定义的参数类型不一致，就会抛出一个可捕获的致命错误。

```
<?php
// 两个类的对象
$myclass = new MyClass;
$otherclass = new OtherClass;

// 致命错误：第一个参数必须是 OtherClass 类的一个对象
$myclass->test('hello');

// 致命错误：第一个参数必须为 OtherClass 类的一个实例
$foo = new stdClass;
$myclass->test($foo);

// 致命错误：第一个参数不能为 null
$myclass->test(null);

// 正确：输出 Hello World
$myclass->test($otherclass);

// 致命错误：第一个参数必须为数组
```

```php
$myclass->test_array('a string');

// 正确：输出数组
$myclass->test_array(array('a', 'b', 'c'));

// 正确：输出 ArrayObject
$myclass->test_interface(new ArrayObject(array()));

// 正确：输出 int(1)
$myclass->test_callable('var_dump', 1);
?>
```

类型约束不只是用在类的成员函数里，也可以使用在函数里。

```php
<?php
// 如下面的类
class MyClass {
    public $var = 'Hello World';
}

/**
 * 测试函数
 * 第一个参数必须是 MyClass 类的一个对象
 */
function MyFunction (MyClass $foo) {
    echo $foo->var;
}

// 正确
$myclass = new MyClass;
MyFunction($myclass);
?>
```

类型约束允许 NULL 值，例如：

```php
<?php
/* 接受 NULL 值 */
function test(stdClass $obj = NULL) {

}

test(NULL);
test(new stdClass);
?>
```

Chapter 80

Late Static Bindings

后期静态绑定（late static binding）功能用于在继承范围内引用静态调用的类。

在引入后期静态绑定之前，测试代码如下：

```
class A{
  static $word = "hello";
  static function hello(){
    print self::$word;
  }
}
class B extends A{
  static $word = "bye";
}
B::hello(); // hello
```

在引入静态绑定之后，测试代码如下：

```
class A{
  static $word = "hello";
  static function hello(){
    print static::$word;
  }
}
class B extends A{
  static $word = "bye";
}
B::hello(); // bye
```

准确地说，后期静态绑定的工作原理是存储了在上一个“非转发调用”（non-forwarding call）的类名。

- 当进行静态方法调用时，该类名即为明确指定的那个（通常在:: 运算符左侧部分）；
- 当进行非静态方法调用时，即为该对象所属的类。

所谓的“转发调用”（forwarding call）指的是通过 `self::`、`parent::`、`static::` 以及 `forward_static_call()` 等方式进行的静态调用，而且可以用 get_called_class() 函数来得到被调用的方法所在的类名，`static::` 则指出了其范围。

上述功能从语言内部角度考虑被命名为“后期静态绑定”，这里的“后期绑定”的意思是说 `static::` 不再被解析为定义当前方法所在的类，而是在实际运行时计算的，因此也可以称之为“静态绑定”，因为它可以用于（但不限于）静态方法的调用。

使用 `self::` 或者 `__CLASS__` 对当前类的静态引用，取决于定义当前方法所在的类。

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        self::who();
    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>
```

以上例程会输出：`A`

后期静态绑定原本的想法是通过引入一个新的关键字表示运行时最初调用的类来绕过限制。简单地说，这个关键字能够在上述例子中调用 test() 时引用的类是 B 而不是 A，最终还是决定不引入新的关键字，而是使用已经预留的 `static::` 关键字。

`static::` 只能用于静态属性。

```
<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
```

```
        static::who(); // 后期静态绑定从这里开始
    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>
```

以上例程会输出：B

在非静态环境下，所调用的类即为该对象实例所属的类，而且 `$this->` 会在同一作用范围内尝试调用私有方法，`static::` 则可能给出不同结果。

```
<?php
class A {
    private function foo() {
        echo "success!\n";
    }
    public function test() {
        $this->foo();
        static::foo();
    }
}

class B extends A {
   /* foo() will be copied to B, hence its scope will still be A and
    * the call be successful */
}

class C extends A {
    private function foo() {
        /* original method is replaced; the scope of the new one is C */
    }
}

$b = new B();
$b->test();
$c = new C();
```

```
$c->test(); //fails
?>
```

以上例程会输出：

```
success!
success!
success!

Fatal error:  Call to private method C::foo() from context 'A' in /tmp/test.php on line 9
```

后期静态绑定的解析会一直到取得一个完全解析了的静态调用为止，如果静态调用使用 `parent::` 或者 `self::` 将转发调用信息。

Example 77 转发和非转发调用

```
<?php
class A {
    public static function foo() {
        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class B extends A {
    public static function test() {
        A::foo();
        parent::foo();
        self::foo();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}
class C extends B {
    public static function who() {
        echo __CLASS__."\n";
    }
```

```
}

C::test();
?>
```

以上例程会输出:

```
A
C
C
```

Chapter 81

References Passing

虽然默认情况下对象是通过引用传递的，但是其实这不是完全正确的。

PHP 的引用是别名，就是两个不同的变量名字指向相同的内容。

一个对象变量已经不再保存整个对象的值，只是保存一个标识符来访问真正的对象内容。

当对象作为参数传递、作为结果返回或者赋值给另外一个变量时，另外一个变量跟原来的不是引用的关系，只是它们都保存着同一个标识符的拷贝，这个标识符指向同一个对象的真正内容。

Example 78　引用和对象

```
<?php
class A {
    public $foo = 1;
}

$a = new A;
$b = $a;    // $a ,$b都是同一个标识符的拷贝
            // ($a) = ($b) = <id>
$b->foo = 2;
echo $a->foo."\n";


$c = new A;
$d = &$c;   // $c ,$d是引用
            // ($c,$d) = <id>

$d->foo = 2;
```

```
echo $c->foo."\n";



$e = new A;

function foo($obj) {
    // ($obj) = ($e) = <id>
    $obj->foo = 2;
}

foo($e);
echo $e->foo."\n";

?>
```

以上例程会输出：

```
2
2
2
```

Chapter 82

Object Serialization

82.1 Sessions Serialization

- serialize() 函数能够将 PHP 中的所有的值转换成一个包含字节流的字符串。
- unserialize() 函数能够重新把字符串变回 PHP 原来的值。

这里序列化一个对象将会保存对象的所有变量，但是不会保存对象的方法，只会保存类的名字。

- 如果序列化类 A 的一个对象，将会返回一个跟类 A 相关，而且包含了对象所有变量值的字符串。
- 如果要想在另外一个文件中解序列化一个对象，这个对象的类必须在解序列化之前定义，可以通过包含一个定义该类的文件或使用函数 spl_autoload_register() 来实现。

为了能够 unserialize() 一个对象，这个对象的类必须已经定义过。

```
<?php
// classa.inc:

  class A {
     public $one = 1;

     public function show_one() {
         echo $this->one;
     }
  }

// page1.php:

  include("classa.inc");
```

```
  $a = new A;
  $s = serialize($a);
  // 把变量$s保存起来以便文件page2.php能够读到
  file_put_contents('store', $s);

// page2.php:

  // 要正确了解序列化，必须包含下面一个文件
  include("classa.inc");

  $s = file_get_contents('store');
  $a = unserialize($s);

  // 现在可以使用对象$a里面的函数 show_one()
  $a->show_one();
?>
```

当一个应用程序使用函数 session_register()[1]来保存对象到会话中时，在每个页面结束的时候这些对象都会自动序列化，并且在每个页面开始的时候又自动解序列化，这样将对象被保存在会话中以后，整个应用程序的页面就都能使用这些对象。

在应用程序中序列化对象以便在之后使用，需要在整个应用程序都包含对象的类的定义，否则有可能出现在解序列化对象的时候，没有找到该对象的类的定义，从而把没有方法的类 __PHP_Incomplete_Class_Name 作为该对象的类，就会返回一个没有用的对象。

当运行 `session_register("a")` 来把变量 $a 放在会话里之后，需要在每个页面都包含文件 classa.inc，而不是只有文件 page1.php 和 page2.php。

[1] session_register() 函数已废弃，不要依赖这个函数。

Part X

Namespaces

Chapter 83

Overview

从广义上来说，命名空间（namespace）是一种封装事物的方法。

命名空间的抽象概念很常见。例如，在操作系统中目录用来将相关文件分组，那么对于目录中的文件来说，它就扮演了命名空间的角色。

举个具体的例子，文件 foo.txt 可以同时在目录/home/greg 和 /home/other 中存在，在同一个目录中不能存在两个 foo.txt 文件，而且在目录 /home/greg 外访问 foo.txt 文件时，就必须将目录名以及目录分隔符放在文件名之前得到 /home/greg/foo.txt。

上述的原理应用到程序设计领域就是命名空间的概念，命名空间用来解决在编写类库或应用程序时创建可重用的代码（如类或函数）时碰到的两类问题：

- 用户编写的代码与 PHP 内部的类/函数/常量或第三方类/函数/常量之间的名字冲突。
- 为过长的标识符名称 (通常是为了缓解第一类问题而定义的) 创建一个别名（或简短）的名称来提高源代码的可读性。

PHP 命名空间提供了一种将相关的类、函数和常量组合到一起的途径。

```
<?php
namespace my\name;

class MyClass {}
function myfunction() {}
const MYCONST = 1;
$a = new MyClass;
$c = new \my\name\MyClass;
$a = strlen('hi');
$d = namespace\MYCONST;
$d = __NAMESPACE__ . '\MYCONST';
echo constant($d);
?>
```

虽然任意合法的 PHP 代码都可以包含在命名空间中，但是只有三种类型的代码受命名空间的影响，它们分别是类（包括抽象类和 traits）、接口、函数和常量。

名字为 PHP 或 php 的命名空间，以及以这些名字开头的命名空间（例如 PHP\Classes）被保留给语言内核使用，不应该在用户空间的代码中使用。

83.1 Namespace

命名空间通过关键字 namespace 来声明。

命名空间类似于操作系统中的目录，两个同名的文件可以共存在不同的目录下，同理两个同名的 PHP 类可以在不同的 PHP 命名空间下共存。

如果一个文件中包含命名空间，那么必须在其它所有代码之前声明命名空间（除了 declare 关键字）。

```
<?php
namespace MyProject;

const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */ }
?>
```

在声明命名空间之前唯一合法的代码是用于定义源文件编码方式的 declare 语句。

另外，所有非 PHP 代码包括空白符都不能出现在命名空间的声明之前。

```
<html>
<?php
// 致命错误 - 命名空间必须是程序脚本的第一条语句
namespace MyProject;
?>
```

与 PHP 其它的语言特征不同，同一个命名空间可以定义在多个文件中，即允许将同一个命名空间的内容分割存放在不同的文件中。

83.2 Sub-namespace

PHP 命名空间允许指定层次化的命名空间的名称，因此命名空间的名字可以使用分层次的方式定义。

```
<?php
namespace MyProject\Sub\Level;
```

```
const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */ }
?>
```

上面的例子创建了常量 MyProject\Sub\Level\CONNECT_OK，类 MyProject\Sub\Level\Connection 和函数 MyProject\Sub\Level\connect。

83.3 Multiple namespace

可以在同一个文件中定义多个命名空间。

在同一个文件中定义多个命名空间有两种语法形式。

```
<?php
namespace MyProject;

const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */ }

namespace AnotherProject;

const CONNECT_OK = 1;
class Connection { /* ... */ }
function connect() { /* ... */ }
?>
```

不过，不建议使用上述语法在单个文件中定义多个命名空间，建议使用下面的大括号形式的语法。

```
<?php
namespace MyProject {
  const CONNECT_OK = 1;
  class Connection { /* ... */ }
  function connect() { /* ... */ }
}

namespace AnotherProject {
  const CONNECT_OK = 1;
  class Connection { /* ... */ }
  function connect() { /* ... */ }
}
?>
```

在实际的编程实践中，不提倡在同一个文件中定义多个命名空间，这种方式的主要用于将多个 PHP 脚本合并在同一个文件中。

将全局的非命名空间中的代码与命名空间中的代码组合在一起，只能使用大括号形式的语法，而且全局代码必须用一个不带名称的 namespace 语句加上大括号括起来，例如：

```
<?php
namespace MyProject {
  const CONNECT_OK = 1;
  class Connection { /* ... */ }
  function connect() { /* ... */ }
}

namespace { // global code
  session_start();
  $a = MyProject\connect();
  echo MyProject\Connection::start();
}
?>
```

除了开始的 declare 语句外，命名空间的括号外不得有任何 PHP 代码。

```
<?php
declare(encoding='UTF-8');
namespace MyProject {
  const CONNECT_OK = 1;
  class Connection { /* ... */ }
  function connect() { /* ... */ }
}

namespace { // 全局代码
  session_start();
  $a = MyProject\connect();
  echo MyProject\Connection::start();
}
?>
```

Chapter 84

Using namespaces

在讨论如何使用命名空间之前，必须了解 PHP 是如何知道要使用哪一个命名空间中的元素的。

这里，将 PHP 命名空间与文件系统作一个简单的类比，文件系统中访问一个文件有三种方式：

1. 相对文件名形式（例如 foo.txt）
   它会被解析为 currentdirectory/foo.txt，其中 currentdirectory 表示当前目录。因此如果当前目录是 /home/foo，则该文件名被解析为/home/foo/foo.txt。
2. 相对路径名形式（例如 subdirectory/foo.txt）
   它会被解析为 currentdirectory/subdirectory/foo.txt。
3. 绝对路径名形式（例如/main/foo.txt）
   它会被解析为/main/foo.txt。

PHP 命名空间中的元素使用同样的原理。例如，类名可以通过三种方式引用：

1. 非限定名称或不包含前缀的类名称（例如 $a=new foo(); 或 foo::staticmethod();）
   如果当前命名空间是 currentnamespace，foo 将被解析为 currentnamespace\foo。如果使用 foo 的代码是全局的，不包含在任何命名空间中的代码，则 foo 会被解析为 foo。
   如果命名空间中的函数或常量未定义，则该非限定的函数名称或常量名称会被解析为全局函数名称或常量名称。
2. 限定名称或包含前缀的名称（例如 $a = new subnamespace\foo(); 或 subnamespace\foo::staticmethod();）
   如果当前的命名空间是 currentnamespace，则 foo 会被解析为 currentnamespace\subnamespace\foo。
   如果使用 foo 的代码是全局的，不包含在任何命名空间中的代码，foo 会被解析为 subnamespace\foo。
3. 完全限定名称或包含了全局前缀操作符的名称（例如 $a = new \currentnamespace\foo(); 或 \currentnamespace\foo::staticmethod();）
   foo 总是被解析为代码中的字面名字 currentnamespace\foo。

```
file1.php
<?php
namespace Foo\Bar\subnamespace;

const FOO = 1;
function foo() {}
class foo
{
    static function staticmethod() {}
}
?>
file2.php
<?php
namespace Foo\Bar;
include 'file1.php';

const FOO = 2;
function foo() {}
class foo
{
    static function staticmethod() {}
}
?>
```

- /* 非限定名称 */

```
foo(); // 解析为 Foo\Bar\foo
foo::staticmethod(); // 解析为类 Foo\Bar\foo的静态方法staticmethod。
echo FOO; // 解析为常量Foo\Bar\FOO
```

- 限定名称

```
subnamespace\foo(); // 解析为函数 Foo\Bar\subnamespace\foo
subnamespace\foo::staticmethod(); // 解析为类 Foo\Bar\subnamespace\foo,
                                  // 以及类的方法 staticmethod
echo subnamespace\FOO; // 解析为常量 Foo\Bar\subnamespace\FOO
```

- 完全限定名称

```
\Foo\Bar\foo(); // 解析为函数 Foo\Bar\foo
\Foo\Bar\foo::staticmethod(); // 解析为类 Foo\Bar\foo, 以及类的方法 staticmethod
echo \Foo\Bar\FOO; // 解析为常量 Foo\Bar\FOO
```

访问任意全局类、函数或常量，都可以使用完全限定名称，例如 \strlen() 或 \Exception 或 \INI_ALL。

Example 79　在命名空间内部访问全局类、函数和常量

```
<?php
namespace Foo;

function strlen() {}
const INI_ALL = 3;
class Exception {}

$a = \strlen('hi'); // 调用全局函数strlen
$b = \INI_ALL; // 访问全局常量 INI_ALL
$c = new \Exception('error'); // 实例化全局类 Exception
?>
```

84.1 Dynamic Language

PHP 命名空间的实现受到其语言自身的动态特征的影响。

Example 80　动态访问元素

```
example1.php:
<?php
class classname
{
    function __construct()
    {
        echo __METHOD__,"\n";
    }
}
function funcname()
{
    echo __FUNCTION__,"\n";
}
const constname = "global";

$a = 'classname';
$obj = new $a; // prints classname::__construct
$b = 'funcname';
$b(); // prints funcname
echo constant('constname'), "\n"; // prints global
?>
```

在进行代码转换时，必须使用完全限定名称（包括命名空间前缀的类名称）。

注意，因为在动态的类名称、函数名称或常量名称中，限定名称和完全限定名称没有区别，因此其前导的反斜杠是不必要的。

Example 81 动态访问命名空间的元素

```
<?php
namespace namespacename;
class classname
{
    function __construct()
    {
        echo __METHOD__,"\n";
    }
}
function funcname()
{
    echo __FUNCTION__,"\n";
}
const constname = "namespaced";

include 'example1.php';

$a = 'classname';
$obj = new $a; // prints classname::__construct
$b = 'funcname';
$b(); // prints funcname
echo constant('constname'), "\n"; // prints global

/* 如果使用双引号，"\\namespacename\\classname"使用双斜杠
$a = '\namespacename\classname';
$obj = new $a; // prints namespacename\classname::__construct
$a = 'namespacename\classname';
$obj = new $a; // also prints namespacename\classname::__construct
$b = 'namespacename\funcname';
$b(); // prints namespacename\funcname
$b = '\namespacename\funcname';
$b(); // also prints namespacename\funcname
echo constant('\namespacename\constname'), "\n"; // prints namespaced
echo constant('namespacename\constname'), "\n"; // also prints namespaced
?>
```

84.2 Nested namespace

PHP 不允许嵌套命名空间。例如，下面的代码是非法的：

Example 82 不允许嵌套命名空间

```
<?php
namespace my\stuff {
   namespace nested {
      class foo {}
   }
}
?>
```

不过，嵌套命名空间的功能可以通过其他方式进行模拟，例如：

Example 83 模拟嵌套命名空间

```
<?php
namespace my\stuff\nested {
   class foo {}
}
?>
```

Chapter 85

Abstract accessing

PHP 支持两种抽象的访问当前命名空间内部元素的方法，__NAMESPACE__ 魔术常量和 namespace 关键字。

85.1 __NAMESPACE__

常量 __NAMESPACE__ 的值是包含当前命名空间名称的字符串。

Example 84 在命名空间中的代码

```
<?php
namespace MyProject;

echo '"', __NAMESPACE__, '"'; // 输出 "MyProject"
?>
```

在全局的，不包括在任何命名空间中的代码，它包含一个空的字符串。

Example 85 全局代码

```
<?php
echo '"', __NAMESPACE__, '"'; // 输出 ""
?>
```

常量 __NAMESPACE__ 在动态创建名称时很有用，例如：

Example 86 使用 __NAMESPACE__ 动态创建名称

```
<?php
namespace MyProject;

function get($classname)
{
    $a = __NAMESPACE__ . '\\' . $classname;
    return new $a;
}
?>
```

85.2 namespace

关键字 namespace 可用来显式访问当前命名空间或子命名空间中的元素，等价于类中的 self 操作符。

Example 87 namespace 操作符

```
<?php
namespace MyProject;

use blah\blah as mine; // see "Using namespaces: importing/aliasing"

blah\mine(); // calls function blah\blah\mine()
namespace\blah\mine(); // calls function MyProject\blah\mine()

namespace\func(); // calls function MyProject\func()
namespace\sub\func(); // calls function MyProject\sub\func()
namespace\cname::method(); // calls static method "method" of class
    MyProject\cname
$a = new namespace\sub\cname(); // instantiates object of class MyProject\
    sub\cname
$b = namespace\CONSTANT; // assigns value of constant MyProject\CONSTANT to
    $b
?>
```

Example 88 全局代码

```
<?php

namespace\func(); // calls function func()
namespace\sub\func(); // calls function sub\func()
```

```
namespace\cname::method(); // calls static method "method" of class cname
$a = new namespace\sub\cname(); // instantiates object of class sub\cname
$b = namespace\CONSTANT; // assigns value of constant CONSTANT to $b
?>
```

Chapter 86

Namespace Aliasing/Importing

命名空间允许通过别名引用或导入外部的完全限定名称，这一点类似于在类 UNIX 文件系统中可以创建对其它的文件或目录的符号连接。

PHP 命名空间支持三种别名或导入方式：为类名称使用别名、为接口使用别名或为命名空间名称使用别名，而且 PHP 允许导入函数或常量或者为它们设置别名。

86.1 use operator

PHP 的别名操作是通过操作符 use 来实现的。

Example 89 使用 use 操作符导入/使用别名

```
<?php
namespace foo;
use My\Full\Classname as Another;

// 下面的例子与 use My\Full\NSname as NSname 相同
use My\Full\NSname;

// 导入一个全局类
use ArrayObject;

// importing a function (PHP 5.6+)
use function My\Full\functionName;

// aliasing a function (PHP 5.6+)
use function My\Full\functionName as func;
```

```
// importing a constant (PHP 5.6+)
use const My\Full\CONSTANT;

$obj = new namespace\Another; // 实例化 foo\Another 对象
$obj = new Another; // 实例化 My\Full\Classname  对象
NSname\subns\func(); // 调用函数 My\Full\NSname\subns\func
$a = new ArrayObject(array(1)); // 实例化 ArrayObject 对象
// 如果不使用 "use \ArrayObject" , 则实例化一个 foo\ArrayObject 对象
func(); // calls function My\Full\functionName
echo CONSTANT; // echoes the value of My\Full\CONSTANT
?>
```

注意，对命名空间中的名称（包含命名空间分隔符的完全限定名称如 Foo\Bar 以及相对的不包含命名空间分隔符的全局名称如 FooBar）来说，前导的反斜杠是不必要的，因为导入的名称必须是完全限定的，不会根据当前的命名空间作相对解析。

PHP 支持在一行中使用多个 use 语句来简化导入/使用别名。

Example 90　一行中包含多个 use 语句

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // 实例化 My\Full\Classname 对象
NSname\subns\func(); // 调用函数 My\Full\NSname\subns\func
?>
```

导入操作是在编译执行的，但是动态的类名称、函数名称或常量名称则不是，因此不受影响。

Example 91　导入和动态名称

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // 实例化一个 My\Full\Classname 对象
$a = 'Another';
$obj = new $a;    // 实际化一个 Another 对象
?>
```

导入操作只影响非限定名称和限定名称。其中，完全限定名称由于是确定的，故不受导入的影响。

Example 92　导入和完全限定名称

```
<?php
use My\Full\Classname as Another, My\Full\NSname;

$obj = new Another; // instantiates object of class My\Full\Classname
$obj = new \Another; // instantiates object of class Another
$obj = new Another\thing; // instantiates object of class My\Full\Classname\
    thing
$obj = new \Another\thing; // instantiates object of class Another\thing
?>
```

user 关键字必须在文件（全局范围）或内部命名空间声明的最外层范围内声明，原因是导入操作是在编译阶段而不是运行时进行的，因此导入语句不能被阻塞。

Example 93　不合法的导入

```
<?php
namespace Languages;

class Greenlandic
{
    use Languages\Danish;
    //...
}
?>
```

导入操作是在每个文件的基础上的，因此包含的文件无法继承父文件导入的名字空间。

86.2 Global spaces

如果没有定义任何命名空间，所有的类与函数的定义都是在全局空间，与 PHP 引入命名空间概念前一样。

```
<?php
$a = new \stdClass;//equals to
$a = new stdClass;
?>
```

在名称前加上前缀 \表示该名称是全局空间中的名称，即使该名称位于其它的命名空间中时也是如此。

Example 94 使用全局空间

```
<?php
namespace A\B\C;

/* 这个函数是 A\B\C\fopen */
function fopen() {
    /* ... */
    $f = \fopen(...); // 调用全局的fopen函数
    return $f;
}
?>
```

86.3 Internal class

```
<?php
namespace foo;
$a = new \stdClass;

function test(\ArrayObject $typehintexample = null) {}

$a = \DirectoryIterator::CURRENT_AS_FILEINFO;

// extending an internal or global class
class MyException extends \Exception {}
?>
```

Chapter 87

Namespace Fallback

在一个命名空间中，当 PHP 遇到一个非限定的类、函数或常量名称时，它使用不同的优先策略来解析该名称。

类名称总是解析到当前命名空间中的名称，因此在访问系统内部或不包含在命名空间中的类名称时，必须使用完全限定名称。

Example 95　在命名空间中访问全局类

```
<?php
namespace A\B\C;
class Exception extends \Exception {}

$a = new Exception('hi'); // $a 是类 A\B\C\Exception 的一个对象
$b = new \Exception('hi'); // $b 是类 Exception 的一个对象

$c = new ArrayObject; // 致命错误，找不到 A\B\C\ArrayObject 类
?>
```

对于函数和常量来说，如果当前命名空间中不存在该函数或常量，PHP 会退而使用全局空间中的函数或常量。

For functions and constants, PHP will fall back to global functions or constants if a namespaced function or constant does not exist.

Example 96　命名空间中后备的全局函数/常量

```
<?php
namespace A\B\C;

const E_ERROR = 45;
```

```
function strlen($str)
{
    return \strlen($str) - 1;
}

echo E_ERROR, "\n"; // 输出 "45"
echo INI_ALL, "\n"; // 输出 "7" - 使用全局常量 INI_ALL

echo strlen('hi'), "\n"; // 输出 "1"
if (is_array('hi')) { // 输出 "is not array"
    echo "is array\n";
} else {
    echo "is not array\n";
}
?>
```

Chapter 88

Namespace Resolution

非限定名称 Unqualified name
名称中不包含命名空间分隔符的标识符，例如 Foo
限定名称 Qualified name
名称中含有命名空间分隔符的标识符，例如 Foo\Bar
完全限定名称 Fully qualified name
名称中包含命名空间分隔符，并以命名空间分隔符开始的标识符，例如 \Foo\Bar，namespace\Foo 也是一个完全限定名称。

名称解析遵循下列规则：

1. 对完全限定名称的函数，类和常量的调用在编译时解析。例如 new \A\B 解析为类 A\B。
2. 所有的非限定名称和限定名称（非完全限定名称）根据当前的导入规则在编译时进行转换。例如，如果命名空间 A\B\C 被导入为 C，那么对 C\D\e() 的调用就会被转换为 A\B\C\D\e()。
3. 在命名空间内部，所有的没有根据导入规则转换的限定名称均会在其前面加上当前的命名空间名称。例如，在命名空间 A\B 内部调用 C\D\e()，则 C\D\e() 会被转换为 A\B\C\D\e() 。
4. 非限定类名根据当前的导入规则在编译时转换（用全名代替短的导入名称）。例如，如果命名空间 A\B\C 导入为 C，则 new C() 被转换为 new A\B\C() 。
5. 在命名空间内部（例如 A\B），对非限定名称的函数调用是在运行时解析的。例如，对函数 foo() 的调用是这样解析的：
   (a) 在当前命名空间中查找名为 A\B\foo() 的函数
   (b) 尝试查找并调用全局 (global) 空间中的函数 foo()。
6. 在命名空间（例如 A\B）内部对非限定名称或限定名称类（非完全限定名称）的调用是在运行时解析的。
   下面是调用 new C() 及 new D\E() 的解析过程：

- new C() 的解析:
  (a) 在当前命名空间中查找 A\B\C 类。
  (b) 尝试自动装载类 A\B\C。
- new D\E() 的解析:
  (a) 在类名称前面加上当前命名空间名称变成：A\B\D\E，然后查找该类。
  (b) 尝试自动装载类 A\B\D\E。

为了引用全局命名空间中的全局类，必须使用完全限定名称 new \C()。

Example 97 名称解析示例

```
<?php
namespace A;
use B\D, C\E as F;

// 函数调用

foo();      // 首先尝试调用定义在命名空间"A"中的函数foo()
            // 再尝试调用全局函数 "foo"

\foo();     // 调用全局空间函数 "foo"

my\foo();   // 调用定义在命名空间"A\my"中函数 "foo"

F();        // 首先尝试调用定义在命名空间"A"中的函数 "F"
            // 再尝试调用全局函数 "F"

// 类引用

new B();    // 创建命名空间 "A" 中定义的类 "B" 的一个对象
            // 如果未找到，则尝试自动装载类 "A\B"

new D();    // 使用导入规则，创建命名空间 "B" 中定义的类 "D" 的一个对象
            // 如果未找到，则尝试自动装载类 "B\D"

new F();    // 使用导入规则，创建命名空间 "C" 中定义的类 "E" 的一个对象
            // 如果未找到，则尝试自动装载类 "C\E"

new \B();   // 创建定义在全局空间中的类 "B" 的一个对象
            // 如果未发现，则尝试自动装载类 "B"

new \D();   // 创建定义在全局空间中的类 "D" 的一个对象
```

```
          // 如果未发现，则尝试自动装载类 "D"

new \F(); // 创建定义在全局空间中的类 "F" 的一个对象
          // 如果未发现，则尝试自动装载类 "F"

// 调用另一个命名空间中的静态方法或命名空间函数

B\foo();  // 调用命名空间 "A\B" 中函数 "foo"

B::foo(); // 调用命名空间 "A" 中定义的类 "B" 的 "foo" 方法
          // 如果未找到类 "A\B" ，则尝试自动装载类 "A\B"

D::foo(); // 使用导入规则，调用命名空间 "B" 中定义的类 "D" 的 "foo" 方法
          // 如果类 "B\D" 未找到，则尝试自动装载类 "B\D"

\B\foo(); // 调用命名空间 "B" 中的函数 "foo"

\B::foo(); // 调用全局空间中的类 "B" 的 "foo" 方法
           // 如果类 "B" 未找到，则尝试自动装载类 "B"

// 当前命名空间中的静态方法或函数

A\B::foo(); // 调用命名空间 "A\A" 中定义的类 "B" 的 "foo" 方法
            // 如果类 "A\A\B" 未找到，则尝试自动装载类 "A\A\B"

\A\B::foo(); // 调用命名空间 "A\B" 中定义的类 "B" 的 "foo" 方法
             // 如果类 "A\B" 未找到，则尝试自动装载类 "A\B"
?>
```

Part XI

Errors

Chapter 89

Overview

PHP 使用内部错误号来报告错误，而且每个 PHP 错误都带有一个类型，PHP 可以根据需要显示和/或记录错误。

如果没有设置错误处理器（error handler），PHP 会根据配置文件来处理错误。例如，php.ini 中的 errror_reporting 配置项说明了哪些错误需要报告，哪些错误需要忽略。

另外，用户还可以使用 error_reporting() 在运行时设置错误报告级别和处理，推荐在 php.ini 中设置错误报告选项，因为有些错误可能发生在 PHP 脚本执行之前，遮掩就可以脚本开始执行之前报告错误。

- 开发阶段建议配置：`E_ALL`
- 生产环境建议配置：`E_ALL & ~E_NOTICE & ~E_STRICT & ~E_DEPRECATED`

实际上，E_ALL 在很多情况下都是推荐的设置，这样就可以提供某些问题的早期预警。

php.ini 中的两个配置选项可以指示如何进一步处理错误：

- display_errors 控制是否把错误显示为 PHP 脚本的输出（在生产环境中应该始终关闭）
- log_errors 控制是否记录错误到日志文件或 syslog（通过 error_log 指令）

如果 PHP 默认的错误处理器不满足需求，用户还可以定制自己的错误处理器来针对不同类型的错误进行处理。例如，在生产环境中可以记录错误并生成错误报告来定位错误。

PHP7 改变了大多数错误的报告方式，不同于传统的错误报告机制，现在大多数错误都被作为 Error 异常抛出，而且 Error 异常可以像 Exception 异常一样被第一个匹配的 try / catch 块所捕获。

- 如果没有匹配的 catch 块，则调用异常处理函数（事先通过 set_exception_handler() 注册）进行处理。
- 如果尚未注册异常处理函数，则按照传统方式处理（即被报告为一个致命错误）。

Error 类并非继承自 Exception 类，所以不能用 `catch (Exception $e) { ... }` 来捕获 Error，而是用 `catch (Error $e) { ... }`，或者通过注册异常处理函数（set_exception_handler()）来捕获 Error。

89.1 Error Handling

在许多「重异常」(exception-heavy) 的编程语言中，一旦发生错误就会抛出异常，而且这也确实是一个可行的方式，不过 PHP 是一个「轻异常」(exception-light) 的语言。

PHP 在处理对象时，其异常机制的核心也会处理问题，只是 PHP 会尽可能的执行而无视发生的事情（除非是一个严重错误）。

PHP 可以忽略 notice 级别的错误并继续执行的特点通常对有「重异常」编程经验的人带来困惑，例如在 Python 中引用一个不存在的变量会抛出异常：

```
$ php -a
php > echo $foo;
Notice: Undefined variable: foo in php shell code on line 1
$ python
>>> print foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'foo' is not defined
```

上述本质上的差异在于 Python 会对任何小错误进行抛错，因此开发人员可以确信任何潜在的问题或者边缘的案例都可以被捕捉到，与此同时 PHP 仍然会保持执行，除非极端的问题发生才会抛出异常。

PHP 定义的错误严重性等级中的三个最常见的的信息类型是错误（error)、通知（notice）和警告（warning），它们有不同的严重性——E_ERROR 、E_NOTICE 和 E_WARNING。

- 错误是运行期间的严重问题，通常是因为代码出错而造成，必须要修正它，否则会使 PHP 停止执行。
- 通知是建议性质的信息，是因为程序代码在执行期有可能造成问题，但程序不会停止。
- 警告是非致命错误，程序执行也不会因此而中止。

另一个在编译期间会报错的信息类型是「E_STRICT」。这个信息用來建议修改程序代码以维持最佳的互通性并能与今后的 PHP 版本兼容。

错误报告可以被 PHP 配置及函数调用改变。其中，使用 PHP 内置的函数 error_reporting() 可以设定程序执行期间的错误等级，方法是传入预定义的错误等级常量，意味着如果只想看到警告和错误（而非通知）时可以这样设定：

```
<?php
error_reporting(E_ERROR | E_WARNING);
```

也可以控制错误是否在屏幕上显示（开发阶段）或隐藏后记录日志（生产环境)。

PHP 提供了错误控制操作符 @ 来抑制特定的错误，只要将这个操作符放置在表达式之前，其后的任何错误都不会出现。

```
<?php
```

```
echo @$foo['bar'];
```

- 如果 $foo['bar'] 存在，程序会将结果输出；
- 如果变量 $foo 或是'bar' 键值不存在，则会返回 null 并且不输出任何东西。

如果不使用错误控制操作符，这个表达式会产生一个错误信息 PHP Notice: Undefined variable: foo 或 PHP Notice: Undefined index: bar 。

使用 @ 操作符需要付出的代价就是 PHP 处理使用 @ 的表达式效率会低一些。过早的性能优化在所有程序语言中也许都是争论点，不过如果性能在应用程序/类库中占有重要地位，那么了解错误控制操作符的性能影响就比较重要。

实际上，错误控制操作符会完全吃掉错误。不但没有显示而且也不会记录在错误日志中。

在正式环境中 PHP 也没有办法关闭错误控制操作符。也许你认为那些错误是无害的，不过那些较具伤害性的错误同时也会被隐藏。

如果有方法可以避免错误抑制符，应该考虑使用。例如，上面的程序代码可以这样重写：

```
<?php
echo isset($foo['bar'] ? $foo['bar'] : '';
```

当 fopen() 载入文件失败时，也许就是一个使用错误抑制符的合理例子。

在尝试载入文件前检查可以是否文件存在，但是如果这个文件在检查后被删除了，此时 fopen() 还未执行（听起来有点不太可能，但是确实会发生），这时 fopen() 会返回 false 并且抛出异常。

虽然上述这种情况也许应该由 PHP 本身来解决，不过这就是使用一个错误抑制符才能有效解决的例子。

虽然在正式的 PHP 环境中没有办法关闭错误控制操作符，但是 Xdebug 有一个 xdebug.scream 的 ini 配置项可以关闭错误控制操作符。例如，可以按照下面的方式修改 php.ini：

```
xdebug.scream = On
```

也可以在执行期间通过 ini_set 函数来设置这个值：

```
<?php
ini_set('xdebug.scream', '1');
```

另外，「Scream」这个 PHP 扩展提供了和 xDebug 类似的功能，只是 Scream 的 ini 设置项叫做 scream.enabled 。

在调试代码而错误信息被隐藏时，这是最有用的方法，不过需要务必小心使用 scream ，应该仅仅是把它当作暂时性的调试工具，例如在许多的 PHP 函数类库代码无法在错误抑制操作符停用时正常使用。

实际上，PHP 本身可以完美化身为「重异常」的程序语言，基本上用户可以利用 ErrorException 类抛出「错误」来当做「异常」，这个类是继承自 Exception 类。

Error/Exception 是大量的现代框架（比如 Symfony 和 Laravel）中的一个常见的做法。例如，Laravel 默认使用 `Whoops!` 扩展包来处理错误，如果 app.debug 启动的话，就会将错误当成异常显示出来，而关闭则会隐藏。

在开发过程中将错误当作异常抛出可以更好的处理它，如果在开发时发生异常，可以将它包在一个 catch 语句中具体说明这种情况如何处理。每捕捉一个异常，都会使应用程序越来越健壮。

以前，PHP 处理异常/错误的问题则比较随意，例如调用 file_get_contents() 函数通常只会给出 FALSE 值和警告，而且许多较早的 PHP 框架（比如 CodeIgniter）只是返回 false，然后将信息写入专有的日志，或者让用户使用类似 `$this->upload->get_error()` 的方法来查看错误原因。这里的问题在于用户必须自己找出错误所在，并且通过翻阅文档查看这个类使用了什么样的错误的方法，而不是明确的暴露错误。

另一个问题发生在当类自动抛出错误到屏幕时会结束程序，这样做阻碍了其他开发者动态处理错误的机会，因此应该抛出异常让开发人员意识到错误的存在，让他们可以选择处理的方式，例如：

```
<?php
$email = new Fuel\Email;
$email->subject('My Subject');
$email->body('How the heck are you?');
$email->to('guy@example.com' , 'Some guy');

try
{
  $email->send();
}
catch (Fuel\Email\ValidationFailedException $e)
{
  // 验证失败
}
catch (Fuel\Email\SendingFailedException $e)
{
  // 驱动问题导致无法发送email
}
finally
{
  // 无论抛出什么样的异常都会执行，并且在正常程序继续之前执行
}
```

89.1.1 error_reporting()

89.1.2 set_error_handler()

虽然某些错误类型无法被处理，不过 PHP 允许定制错误处理的方式。例如，可以向用户显示一个自定义错误页面，也可以通过发送电子邮件而不是日志的方式来报告错误。

89.2 Error Hierarchy

- Throwable
  - Error
    - ArithmeticError
      - DivisionByZeroError
    - AssertionError
    - ParseError
    - TypeError
  - Exception

Chapter 90

Error

Error 类是 PHP 中的所有的内部错误类的基类，无法克隆。

90.1 Class synopsis

```
Error implements Throwable {
  /* Properties */
  protected string $message ;
  protected int $code ;
  protected string $file ;
  protected int $line ;
  /* Methods */
  public __construct ([ string $message = "" [, int $code = 0 [, Throwable $previous =
      NULL ]]] )
  final public string getMessage ( void )
  final public Throwable getPrevious ( void )
  final public mixed getCode ( void )
  final public string getFile ( void )
  final public int getLine ( void )
  final public array getTrace ( void )
  final public string getTraceAsString ( void )
  public string __toString ( void )
  final private void __clone ( void )
}
```

90.2 Class property

90.2.1 message

错误信息

90.2.2 code

错误代码

90.2.3 file

错误发生的文件

90.2.4 line

错误发生的行数

90.3 Class method

90.3.1 Error::__construct()

```
public Error::__construct ([ string $message = "" [, int $code = 0 [, Throwable
    $previous = NULL ]]] )
```

构造函数

- $message 错误信息
- $code 错误代码
- $previous 异常链上的上一个可抛出错误

$message 不是二进制安全的。

90.3.2 Error::getMessage()

```
final public string Error::getMessage ( void )
```

获取错误信息

```
<?php
try {
   throw new Error("Some error message");
} catch(Error $e) {
   echo $e->getMessage();
```

```
}
```

90.3.3 Error::getPrivious()

```
final public Throwable Error::getPrevious ( void )
```

获取上一个可抛出（Throwable）的的错误（或 NULL）

```
<?php
class MyCustomError extends Error {}

function doStuff() {
    try {
        throw new InvalidArgumentError("You are doing it wrong!", 112);
    } catch(Error $e) {
        throw new MyCustomError("Something happened", 911, $e);
    }
}


try {
    doStuff();
} catch(Error $e) {
    do {
        printf("%s:%d %s (%d) [%s]\n", $e->getFile(), $e->getLine(), $e->getMessage(), $e
            ->getCode(), get_class($e));
    } while($e = $e->getPrevious());
}
?>
```

上述示例的结果如下：

```
/home/me/test.php:8 Something happened (911) [MyCustomError]
/home/me/test.php:6 You are doing it wrong! (112) [InvalidArgumentError]
```

90.3.4 Error::getCode()

```
final public mixed Error::getCode ( void )
```

获取错误代码

```
<?php
try {
```

```
    throw new Error("Some error message", 30);
} catch(Error $e) {
    echo "The Error code is: " . $e->getCode();
}
```

上述示例的结果如下:

```
The Error code is: 30
```

90.3.5 Error::getFile()

```
final public string Error::getFile ( void )
```

获取错误脚本文件名

```
<?php
try {
    throw new Error;
} catch(Error $e) {
    echo $e->getFile();
```

上述示例的结果如下:

```
/home/me/test.php
```

90.3.6 Error::getLine()

```
final public int Error::getLine ( void )
```

获取错误行号

```
<?php
try {
    throw new Error("Some error message");
} catch(Error $e) {
    echo "The error was created on line: " . $e->getLine();
}
```

上述示例的结果如下:

```
The error was created on line: 3
```

90.3.7 Error::getTrace()

```
final public array Error::getTrace ( void )
```

获取堆栈追踪数组信息

```
<?php
function test() {
 throw new Error;
}

try {
 test();
} catch(Error $e) {
 var_dump($e->getTrace());
}
```

上述示例的结果如下:

```
array(1) {
  [0]=>
  array(4) {
    ["file"]=>
    string(22) "/home/me/test.php"
    ["line"]=>
    int(7)
    ["function"]=>
    string(4) "test"
    ["args"]=>
    array(0) {
    }
  }
}
```

90.3.8 Error::getTraceAsString()

```
final public string Error::getTraceAsString ( void )
```

获取字符串表示的堆栈追踪信息

```
<?php
function test() {
    throw new Error;
}
```

```
try {
    test();
} catch(Error $e) {
    echo $e->getTraceAsString();
}
```

上述示例的结果如下：

```
#0 /home/me/test.php(7): test()
#1 {main}
```

90.3.9 Error::__toString()

```
public string Error::__toString ( void )
```

错误的字符串表示

```
<?php
try {
    throw new Error("Some error message");
} catch(Error $e) {
    echo $e;
}
```

上述示例的结果如下：

```
Error: Some error message in /home/me/test.php:3
Stack trace:
#0 {main}
```

90.3.10 Error::__clone()

```
final private void Error::__clone ( void )
```

克隆错误（Error 无法被克隆，因此该方法会导致致命错误）

90.4 Class Interface

90.4.1 Throwable

90.5 Class extends

90.5.1 ArithmeticError

In PHP 7.0, these errors include attempting to perform a bitshift by a negative amount, and any call to intdiv() that would result in a value outside the possible bounds of an integer.

```
ArithmeticError extends Error {
/* Inherited methods */
final public string Error::getMessage ( void )
final public Throwable Error::getPrevious ( void )
final public mixed Error::getCode ( void )
final public string Error::getFile ( void )
final public int Error::getLine ( void )
final public array Error::getTrace ( void )
final public string Error::getTraceAsString ( void )
public string Error::__toString ( void )
final private void Error::__clone ( void )
}
```

算术运算时发生错误会抛出 ArithmeticError。

90.5.2 DivisionByZeroError

```
DivisionByZeroError extends ArithmeticError {
/* Inherited methods */
final public string Error::getMessage ( void )
final public Throwable Error::getPrevious ( void )
final public mixed Error::getCode ( void )
final public string Error::getFile ( void )
final public int Error::getLine ( void )
final public array Error::getTrace ( void )
final public string Error::getTraceAsString ( void )
public string Error::__toString ( void )
final private void Error::__clone ( void )
}
```

试图进行除 0 操作时会抛出除 0 错误。

90.5.3 AssertionError

```
AssertionError extends Error {
/* Inherited methods */
final public string Error::getMessage ( void )
final public Throwable Error::getPrevious ( void )
final public mixed Error::getCode ( void )
final public string Error::getFile ( void )
final public int Error::getLine ( void )
final public array Error::getTrace ( void )
final public string Error::getTraceAsString ( void )
public string Error::__toString ( void )
final private void Error::__clone ( void )
}
```

执行 assert() 发生断言错误时会抛出 AssertionError。

90.5.4 ParseError

```
ParseError extends Error {
/* Inherited methods */
final public string Error::getMessage ( void )
final public Throwable Error::getPrevious ( void )
final public mixed Error::getCode ( void )
final public string Error::getFile ( void )
final public int Error::getLine ( void )
final public array Error::getTrace ( void )
final public string Error::getTraceAsString ( void )
public string Error::__toString ( void )
final private void Error::__clone ( void )
}
```

在解析 PHP 代码（例如执行 eval()）出错会抛出 ParseError。

90.5.5 TypeError

```
TypeError extends Error {
/* Inherited methods */
final public string Error::getMessage ( void )
final public Throwable Error::getPrevious ( void )
final public mixed Error::getCode ( void )
final public string Error::getFile ( void )
```

```
final public int Error::getLine ( void )
final public array Error::getTrace ( void )
final public string Error::getTraceAsString ( void )
public string Error::__toString ( void )
final private void Error::__clone ( void )
}
```

有三种情况会抛出 TypeError，分别是：

- 传递给函数的参数与其对应的声明参数类型不匹配
- 从函数返回的值与声明的函数返回类型不匹配
- 将无效数量的参数传递给 PHP 内置函数（仅限严格模式）

Part XII

Exceptions

Chapter 91

Overview

异常（Exception）用于在指定的错误发生时改变脚本的正常流程。

异常是许多流行编程语言的标配，但它们往往被 PHP 开发人员所忽视，而 Ruby 就是一个极度重视异常的语言，无论有什么错误发生（例如 HTTP 请求失败或者数据库查询有问题，甚至找不到一个图片资源），Ruby（或是所使用的 gems）将会抛出异常，用户可以通过屏幕立刻知道所发生的问题。

一般情况下，PHP 内置的函数使用错误报告来处理异常，只有使用 PHP 进行面向对象编程时才会使用异常处理，不过 PHP 现在已经可以简单地把错误转换为异常并抛出。

注意，Exception 对象不能被复制，尝试对 Exception 对象复制会导致一个 E_ERROR 级别的错误。

91.1 Basic Exception

当一个异常被抛出时，其后的代码（指抛出异常时所在的代码块）不会继续执行，而 PHP 会尝试查找第一个能与之匹配的 “catch” 代码块。

如果一个异常没有被捕获，而且又没有使用 set_exception_handler() 作相应的处理的话，那么 PHP 将会产生一个严重的错误（致命错误），并且输出 “Uncaught Exception ...”（未捕获异常）的错误提示信息。

在下面的示例中，尝试抛出一个异常，同时不去捕获它：

```
<?php
//create function with an exception
function checkNum($number)
 {
 if($number>1)
  {
```

```
   throw new Exception("Value must be 1 or below");
 }
return true;
}

//trigger exception
checkNum(2);
?>
```

上面的代码会获得类似这样的一个错误：

```
Fatal error: Uncaught exception 'Exception'
with message 'Value must be 1 or below' in test.php:6
Stack trace: #0 test.php(12):
checkNum(28) #1 {main} thrown in test.php on line 6
```

Example 98 抛出一个异常

```
<?php
function inverse($x) {
    if (!$x) {
        throw new Exception('Division by zero.');
    }
    else return 1/$x;
}

try {
    echo inverse(5) . "\n";
    echo inverse(0) . "\n";
} catch (Exception $e) {
    echo 'Caught exception: ', $e->getMessage(), "\n";
}

// Continue execution
echo 'Hello World';
?>
```

以上例程会输出：

```
0.2
Caught exception: Division by zero.
Hello World
```

Example 99 嵌套的异常

```
<?php
class MyException extends Exception { }

class Test {
    public function testing() {
        try {
            try {
                throw new MyException('foo!');
            } catch (MyException $e) {
                /* rethrow it */
                throw $e;
            }
        } catch (Exception $e) {
            var_dump($e->getMessage());
        }
    }
}

$foo = new Test;
$foo->testing();

?>
```

以上例程会输出:

```
string(4) "foo!"
```

91.2 try/throw/catch

要避免上面例子出现的错误，我们需要创建适当的代码来处理异常。

异常处理程序应当包括:

1. try - 使用异常的函数应该位于”try” 代码块内。如果没有触发异常，则代码将照常继续执行。但是如果异常被触发，会抛出一个异常。
2. throw - 这里规定如何触发异常。每一个”throw” 必须对应至少一个”catch”
3. catch - ”catch” 代码块会捕获异常，并创建一个包含异常信息的对象

下面的示例将触发一个异常:

```
<?php
//创建可抛出一个异常的函数
```

```
function checkNum($number)
 {
 if($number>1)
  {
    throw new Exception("Value must be 1 or below");
  }
 return true;
 }

//在 "try" 代码块中触发异常
try
 {
 checkNum(2);
 //If the exception is thrown, this text will not be shown
 echo 'If you see this, the number is 1 or below';
 }

//捕获异常
catch(Exception $e)
 {
 echo 'Message: ' .$e->getMessage();
 }
?>
```

上面代码将获得类似这样一个错误：`Message: Value must be 1 or below`

上面的代码抛出了一个异常，并捕获了它：

1. 创建 checkNum() 函数。它检测数字是否大于 1。如果是，则抛出一个异常。
2. 在 “try” 代码块中调用 checkNum() 函数。
3. checkNum() 函数中的异常被抛出
4. “catch” 代码块接收到该异常，并创建一个包含异常信息的对象 (`$e`)。
5. 通过从这个 exception 对象调用 `$e->getMessage()`，输出来自该异常的错误消息

不过，为了遵循 “每个 throw 必须对应一个 catch” 的原则，可以设置一个顶层的异常处理器来处理漏掉的错误。

91.3 Build-in Exception

PHP 的 SPL（Standard PHP Library）提供了一系列的内置异常类。

- LogicException (extends Exception)
  - BadFunctionException

        * BadMethodException
    - DomainException
    - InvalidArgumentException
    - LengthException
    - OutOfRangeException
- RuntimeException (extends Exception)
    - OutOfBoundsException
    - OverflowException
    - RangeException
    - UnderflowException
    - UnexpectedValueException

91.3.1 BadFunctionCallException

91.3.2 BadMethodCallException

91.3.3 DomainException

91.3.4 InvalidArgumentException

91.3.5 LengthException

91.3.6 LogicException

91.3.7 OutOfBoundsException

91.3.8 OutOfRangeException

91.3.9 OverflowException

91.3.10 RangeException

91.3.11 RuntimeException

91.3.12 UnderflowException

91.3.13 UnexpectedValueException

91.4 Custom Exception

以下这段代码只为说明内置异常处理类的结构，它并不是一段有实际意义的可用代码。

Example 100 内置的异常处理类

```
<?php
class Exception
{
    protected $message = 'Unknown exception'; // 异常信息
    protected $code = 0;                      // 用户自定义异常代码
    protected $file;                          // 发生异常的文件名
    protected $line;                          // 发生异常的代码行号

    function __construct($message = null, $code = 0);

    final function getMessage();              // 返回异常信息
    final function getCode();                 // 返回异常代码
    final function getFile();                 // 返回发生异常的文件名
    final function getLine();                 // 返回发生异常的代码行号
    final function getTrace();                // backtrace() 数组
    final function getTraceAsString();        // 已格成化成字符串的 getTrace() 信
        息

    /* 可重载的方法 */
    function __toString();                     // 可输出的字符串
}
?>
```

用户可以用自定义的异常处理类来扩展 PHP 内置的异常处理类，这样就可以在发生异常时调用自定义 exception 类的函数，只要自定义异常类是 Exception 类的一个扩展。

以下的代码说明了在内置的异常处理类中，哪些属性和方法在子类中是可访问和可继承的。

这个自定义的 exception 类继承了 PHP 的 Exception 类的所有属性，可向其添加自定义的函数。

Example 101 创建 exception 类

```
<?php
class customException extends Exception
 {
 public function errorMessage()
  {
  //error message
  $errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
  .': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
  return $errorMsg;
```

```
 }
}

$email = "someone@example...com";

try
{
//check if
if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
 {
 //throw exception if email is not valid
 throw new customException($email);
 }
}

catch (customException $e)
{
//display custom message
echo $e->errorMessage();
}
?>
```

这个新的类是旧的 Exception 类的副本，外加 errorMessage() 函数。正因为它是旧类的副本，因此它从旧类继承了属性和方法，我们可以使用 Exception 类的方法，比如 getLine()、getFile() 以及 getMessage()。

上面的代码抛出了一个异常，并通过一个自定义的 exception 类来捕获它：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法。
2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一条错误消息
3. 把 $email 变量设置为不合法的 e-mail 地址字符串
4. 执行”try” 代码块，由于 e-mail 地址不合法，因此抛出一个异常
5. ”catch” 代码块捕获异常，并显示错误消息

如果使用自定义的类来扩展内置异常处理类，并且要重新定义构造函数的话，建议同时调用 parent::__construct() 来检查所有的变量是否已被赋值。

当对象要输出字符串的时候，可以重载 __toString() 并自定义输出的样式。

Example 102　扩展 PHP 内置的异常处理类

```
<?php
/**
```

```
 * 自定义一个异常处理类
 */
class MyException extends Exception
{
    // 重定义构造器使 message 变为必须被指定的属性
    public function __construct($message, $code = 0) {
        // 自定义的代码

        // 确保所有变量都被正确赋值
        parent::__construct($message, $code);
    }

    // 自定义字符串输出的样式
    public function __toString() {
        return __CLASS__ . ": [{$this->code}]: {$this->message}\n";
    }

    public function customFunction() {
        echo "A Custom function for this type of exception\n";
    }
}


/**
 * 创建一个用于测试异常处理机制的类
 */
class TestException
{
    public $var;

    const THROW_NONE = 0;
    const THROW_CUSTOM = 1;
    const THROW_DEFAULT = 2;

    function __construct($avalue = self::THROW_NONE) {

        switch ($avalue) {
            case self::THROW_CUSTOM:
                // 抛出自定义异常
                throw new MyException('1 is an invalid parameter', 5);
                break;
```

```
            case self::THROW_DEFAULT:
                // 抛出默认的异常
                throw new Exception('2 isnt allowed as a parameter', 6);
                break;

            default:
                // 没有异常的情况下，创建一个对象
                $this->var = $avalue;
                break;
        }
    }
}


// 例子 1
try {
    $o = new TestException(TestException::THROW_CUSTOM);
} catch (MyException $e) { // 捕获异常
    echo "Caught my exception\n", $e;
    $e->customFunction();
} catch (Exception $e) {    // 被忽略
    echo "Caught Default Exception\n", $e;
}

// 执行后续代码
var_dump($o);
echo "\n\n";


// 例子 2
try {
    $o = new TestException(TestException::THROW_DEFAULT);
} catch (MyException $e) { // 不能匹配异常的种类，被忽略
    echo "Caught my exception\n", $e;
    $e->customFunction();
} catch (Exception $e) {    // 捕获异常
    echo "Caught Default Exception\n", $e;
}

// 执行后续代码
```

```
var_dump($o);
echo "\n\n";


// 例子 3
try {
    $o = new TestException(TestException::THROW_CUSTOM);
} catch (Exception $e) {    // 捕获异常
    echo "Default Exception caught\n", $e;
}

// 执行后续代码
var_dump($o);
echo "\n\n";


// 例子 4
try {
    $o = new TestException();
} catch (Exception $e) {    // 没有异常，被忽略
    echo "Default Exception caught\n", $e;
}

// 执行后续代码
var_dump($o);
echo "\n\n";
?>
```

91.5 Exception Handler

91.5.1 catch

PHP 本身实际上提供了类似于其它面向对象的异常处理模块，例如异常可以被抛出（throw），也可以被抓住（catch），但是 PHP 内部函数主要使用错误报告，只有现代面向对象的扩展才可能使用异常。

- 异常检测被包含在 try 代码块中来捕捉潜在的异常。
- 每个 try 块必须至少有一个对应的 catch 块或 finally 块。

PHP 错误可以很容易的通过 ErrorException 转换为异常，而且 PHP 标准库 (SPL) 提供了许多内建的异常类。

实际上，可以被抛出的对象必须是异常类及其子类的实例，如果试图抛出一个正常对象会导致一个致命错误。

当异常被触发时，通常会发生的情况如下：

- 当前代码状态被保存
- 代码执行被切换到预定义的异常处理器函数
- 根据情况，处理器也许会从保存的代码状态重新开始执行代码，终止脚本执行，或从代码中另外的位置继续执行脚本

在 PHP 代码中所产生的异常可被 throw 语句抛出并被 catch 语句捕获。需要进行异常处理的代码都必须放入 try 代码块内，以便捕获可能存在的异常。

- 每一个 try 至少要有一个与之对应的 catch。
- 使用多个 catch 可以捕获不同的异常类实例。

当 try 代码块不再抛出异常或者找不到 catch 能匹配所抛出的异常时，PHP 代码就会在跳转到最后一个 catch 的后面继续执行，而且 PHP 允许在 catch 代码块内再次抛出（throw）异常。

91.5.2 finally

finally 块可以定义在 catch 块后面或者代替 catch 块，这样 catch 块中的代码将始终会被执行，无论是否已抛出异常，而且正常执行流程恢复之前也会执行 finally 块中的代码。

```
<?php
function inverse($x) {
    if (!$x) {
        throw new Exception('Division by zero.');
    }
    return 1/$x;
}

try {
    echo inverse(5) . "\n";
} catch (Exception $e) {
    echo 'Caught exception: ', $e->getMessage(), "\n";
} finally {
    echo "First finally.\n";
}

try {
    echo inverse(0) . "\n";
} catch (Exception $e) {
    echo 'Caught exception: ', $e->getMessage(), "\n";
```

```
} finally {
    echo "Second finally.\n";
}

// Continue execution
echo "Hello World\n";
```

Example 103 嵌套的异常处理

```
<?php

class MyException extends Exception { }

class Test {
    public function testing() {
        try {
            try {
                throw new MyException('foo!');
            } catch (MyException $e) {
                // rethrow it
                throw $e;
            }
        } catch (Exception $e) {
            var_dump($e->getMessage());
        }
    }
}

$foo = new Test;
$foo->testing();
```

上述示例会输出：

```
<?php
string(4) "foo!"
```

91.6 Multiple Exceptions

可以为一段脚本使用多个异常，来检测多种情况。

可以使用多个 if..else 代码块，或一个 switch 代码块，或者嵌套多个异常。这些异常能够使用不同的 exception 类，并返回不同的错误消息：

```
<?php
class customException extends Exception
{
public function errorMessage()
{
//error message
$errorMsg = 'Error on line '.$this->getLine().' in '.$this->getFile()
.': <b>'.$this->getMessage().'</b> is not a valid E-Mail address';
return $errorMsg;
}
}

$email = "someone@example.com";

try
 {
 //check if
 if(filter_var($email, FILTER_VALIDATE_EMAIL) === FALSE)
  {
  //throw exception if email is not valid
  throw new customException($email);
  }
 //check for "example" in mail address
 if(strpos($email, "example") !== FALSE)
  {
  throw new Exception("$email is an example e-mail");
  }
 }

catch (customException $e)
 {
 echo $e->errorMessage();
 }

catch(Exception $e)
 {
 echo $e->getMessage();
 }
?>
```

上面的代码测试了两种条件，如何任何条件不成立，则抛出一个异常：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧

类的所有属性和方法。

2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一个错误消息。
3. 执行”try” 代码块，在第一个条件下，不会抛出异常。
4. 由于 e-mail 含有字符串”example”，第二个条件会触发异常。
5. ”catch” 代码块会捕获异常，并显示恰当的错误消息

如果没有捕获 customException，紧紧捕获了 base exception，则在那里处理异常。

91.7 Re-throwing Exceptions

有时，当异常被抛出时，开发者也许希望以不同于标准的方式对它进行处理，因此可以在一个”catch” 代码块中再次抛出异常。

对程序员来说，系统错误也许很重要，但是用户对它们并不感兴趣，因此脚本应该对用户隐藏系统错误。

为了让用户更容易使用，开发者可以再次抛出带有对用户比较友好的消息的异常：

```
<?php
class customException extends Exception
 {
 public function errorMessage()
  {
  //error message
  $errorMsg = $this->getMessage().' is not a valid E-Mail address.';
  return $errorMsg;
  }
 }

$email = "someone@example.com";

try
 {
 try
  {
  //check for "example" in mail address
  if(strpos($email, "example") !== FALSE)
   {
   //throw exception if email is not valid
   throw new Exception($email);
   }
  }
 catch(Exception $e)
```

```
 {
 //re-throw exception
 throw new customException($email);
 }
}

catch (customException $e)
{
//display custom message
echo $e->errorMessage();
}
?>
```

上面的代码检测在邮件地址中是否含有字符串”example”。如果有，则再次抛出异常：

1. customException() 类是作为旧的 exception 类的一个扩展来创建的。这样它就继承了旧类的所有属性和方法。
2. 创建 errorMessage() 函数。如果 e-mail 地址不合法，则该函数返回一个错误消息。
3. 把 $email 变量设置为一个有效的邮件地址，但含有字符串 “example”。
4. “try” 代码块包含另一个 “try” 代码块，这样就可以再次抛出异常。
5. 由于 e-mail 包含字符串 “example”，因此触发异常。
6. “catch” 捕获到该异常，并重新抛出 “customException”。
7. 捕获到 “customException”，并显示一条错误消息。

如果在其目前的 “try” 代码块中异常没有被捕获，则它将在更高层级上查找 catch 代码块。

91.8 Top-level Exception Handler

set_exception_handler() 函数可设置处理所有未捕获异常的用户定义函数。

```
<?php
function myException($exception)
{
echo "<b>Exception:</b> " , $exception->getMessage();
}

set_exception_handler('myException');

throw new Exception('Uncaught Exception occurred');
?>
```

以上代码的输出应该类似这样：`Exception:Uncaught Exception ocuured`

在上面的代码中，不存在"catch" 代码块，而是触发顶层的异常处理程序。应该使用此函数来捕获所有未被捕获的异常。

91.9 Rules for Exception

异常处理的规则包括：

- 需要进行异常处理的代码应该放入 try 代码块内，以便捕获潜在的异常。
- 每个 try 或 throw 代码块必须至少拥有一个对应的 catch 代码块。
- 使用多个 catch 代码块可以捕获不同种类的异常。
- 可以在 try 代码块内的 catch 代码块中再次抛出（re-thrown）异常。

简而言之就是，如果抛出了异常，就必须捕获它。

Chapter 92

Exception

Exception 是所有异常的基类，不过原生的 Exception 类并没有提供太多的调试情境给开发人员，不过可以通过建立一个特殊的 Exception 来弥补它，也就是建立一个继承自原生 Exception 类的一个子类，例如：

```
<?php
class ValidationException extends Exception {}
```

扩展原生的 Exception 类可以加入多个 catch 区块，并且根据不同的异常分别处理，这样就可以建立许多自定义异常，而且其中有些已经在 SPL 扩展提供的 SPL 异常中定义了。

举例来说，如果使用了 __call() 魔术方法去调用一个无效的方法而不是抛出一个模糊的标准 Exception 或是建立自定义的异常处理，就可以直接抛出自定义异常，例如：

```
<?php
throw new BadMethodCallException;
```

92.1 Summary

```
Exception {
  /* 属性 */
  protected string $message ;
  protected int $code ;
  protected string $file ;
  protected int $line ;
  /* 方法 */
  public __construct ([ string $message = "" [, int $code = 0 [, Exception $previous =
      NULL ]]] )
  final public string getMessage ( void )
```

```
  final public Exception getPrevious ( void )
  final public int getCode ( void )
  final public string getFile ( void )
  final public int getLine ( void )
  final public array getTrace ( void )
  final public string getTraceAsString ( void )
  public string __toString ( void )
  final private void __clone ( void )
}
```

92.2 Property

- message 异常消息内容
- code 异常代码
- file 抛出异常的文件名
- line 抛出异常在该文件中的行号

92.3 Method

- Exception::__construct —异常构造函数
- Exception::getMessage —获取异常消息内容
- Exception::getPrevious —返回异常链中的前一个异常
- Exception::getCode —获取异常代码
- Exception::getFile —获取发生异常的程序文件名称
- Exception::getLine —获取发生异常的代码在文件中的行号
- Exception::getTrace —获取异常追踪信息
- Exception::getTraceAsString —获取字符串类型的异常追踪信息
- Exception::__toString —将异常对象转换为字符串
- Exception::__clone —异常克隆

92.3.1 Exception::__construct

Exception::__construct ——异常构造函数[1]。

```
public Exception::__construct() ([ string $message = "" [, int $code = 0 [, Exception
    $previous = NULL ]]] )
```

[1] 自 PHP 5.3.0 起，增加 previous 参数。

参数

- message 抛出的异常消息内容，The message is NOT binary safe.
- code 异常代码。
- previous 异常链中的前一个异常。

92.3.2 Exception::getMessage

Exception::getMessage —获取异常消息内容，返回字符串类型的异常消息内容。此函数没有参数。

```
final public string Exception::getMessage ( void )
```

Example 104 Exception::getMessage() 示例

```
<?php
try {
    throw new Exception("Some error message");
} catch(Exception $e) {
    echo $e->getMessage();
}
?>
```

以上例程的输出类似于:

```
Some error message
```

92.3.3 Exception::getPrevious

Exception::getPrevious —返回异常链中的前一个异常（Exception::__construct() 方法的第三个参数），否则返回 NULL。此函数没有参数。

```
final public Exception Exception::getPrevious ( void )
```

Example 105 Exception::getPrevious() 示例 - 追踪异常，并循环打印。

```
<?php
class MyCustomException extends Exception {}

function doStuff() {
    try {
        throw new InvalidArgumentException("You are doing it wrong!", 112);
    } catch(Exception $e) {
```

```
        throw new MyCustomException("Something happend", 911, $e);
    }
}


try {
    doStuff();
} catch(Exception $e) {
    do {
        printf("%s:%d %s (%d) [%s]\n", $e->getFile(), $e->getLine(), $e->
            getMessage(), $e->getCode(), get_class($e));
    } while($e = $e->getPrevious());
}
?>
```

以上例程的输出类似于:

```
/home/bjori/ex.php:8 Something happend (911) [MyCustomException]
/home/bjori/ex.php:6 You are doing it wrong! (112) [InvalidArgumentException]
```

92.3.4 Exception::getCode

Exception::getCode —获取并返回整型（integer）类型的异常代码，此函数没有参数。

```
final public int Exception::getCode ( void )
```

Example 106 Exception::getCode() 示例

```
<?php
try {
    throw new Exception("Some error message", 30);
} catch(Exception $e) {
    echo "The exception code is: " . $e->getCode();
}
?>
```

以上例程的输出类似于:

```
The exception code is: 30
```

92.3.5 Exception::getFile

Exception::getFile —获取并返回发生异常的程序文件名称，此函数没有参数。

```
final public string Exception::getFile ( void )
```

Example 107 Exception::getFile() 示例

```
<?php
try {
    throw new Exception;
} catch(Exception $e) {
    echo $e->getFile();
}
?>
```

以上例程的输出类似于：

```
/home/bjori/tmp/ex.php
```

92.3.6 Exception::getLine

Exception::getLine —获取并返回发生异常的代码在文件中的行号，此函数没有参数。

```
final public int Exception::getLine ( void )
```

Example 108 Exception::getLine() 示例

```
<?php
try {
    throw new Exception("Some error message");
} catch(Exception $e) {
    echo "The exception was thrown on line: " . $e->getLine();
}
?>
```

以上例程的输出类似于：

```
The exception was thrown on line: 3
```

92.3.7 Exception::getTrace

Exception::getTrace —获取并返回异常追踪信息的数组（array），此函数没有参数。

```
final public array Exception::getTrace ( void )
```

Example 109 Exception::getTrace() 示例

```
<?php
function test() {
 throw new Exception;
}

try {
 test();
} catch(Exception $e) {
 var_dump($e->getTrace());
}
?>
```

以上例程的输出类似于：

```
array(1) {
  [0]=>
  array(4) {
    ["file"]=>
    string(22) "/home/bjori/tmp/ex.php"
    ["line"]=>
    int(7)
    ["function"]=>
    string(4) "test"
    ["args"]=>
    array(0) {
    }
  }
}
```

92.3.8 Exception::getTraceAsString

Exception::getTraceAsString —获取并返回字符串类型的异常追踪信息。此函数没有参数。

```
final public string Exception::getTraceAsString ( void )
```

Example 110 Exception::getTraceAsString() 示例

```
<?php
function test() {
    throw new Exception;
}

try {
    test();
} catch(Exception $e) {
    echo $e->getTraceAsString();
}
?>
```

以上例程的输出类似于:

```
#0 /home/bjori/tmp/ex.php(7): test()
#1 {main}
```

92.3.9 Exception::__toString

Exception::__toString 一将异常对象转换为字符串并返回。此函数没有参数。

```
public string Exception::__toString ( void )
```

Example 111 Exception::__toString() 示例

```
<?php
try {
    throw new Exception("Some error message");
} catch(Exception $e) {
    echo $e;
}
?>
```

以上例程的输出类似于:

```
exception 'Exception' with message 'Some error message' in /home/bjori/tmp/ex.php:3
Stack trace:
#0 {main}
```

92.3.10 Exception::__clone

Exception::__clone 一尝试异常克隆。因为异常被不允许克隆，因而这将导致一个致命错误。此函数没有参数，没有返回值。

```
final private void Exception::__clone ( void )
```

Chapter 93

ErrorException

错误异常。

93.1 Summary

```
ErrorException extends Exception {
  /* 属性 */
   protected int $severity ;
  /* 方法 */
  public __construct ([ string $message = "" [, int $code = 0 [, int $severity = 1 [,
      string $filename = __FILE__ [, int $lineno = __LINE__ [, Exception $previous =
      NULL ]]]]]] )
  final public int getSeverity ( void )
  /* 继承的方法 */
  final public string Exception::getMessage ( void )
  final public Exception Exception::getPrevious ( void )
  final public int Exception::getCode ( void )
  final public string Exception::getFile ( void )
  final public int Exception::getLine ( void )
  final public array Exception::getTrace ( void )
  final public string Exception::getTraceAsString ( void )
  public string Exception::__toString ( void )
  final private void Exception::__clone ( void )
}
```

93.2 Property

- severity 异常级别

93.3 Method

- ErrorException::__construct —异常构造函数
- ErrorException::getSeverity —获取异常的严重程度

Example 112 使用 set_error_handler() 函数将错误信息托管至 ErrorException

```
<?php
function exception_error_handler($errno, $errstr, $errfile, $errline ) {
    throw new ErrorException($errstr, 0, $errno, $errfile, $errline);
}
set_error_handler("exception_error_handler");

/* Trigger exception */
strpos();
?>
```

以上例程的输出类似于:

```
Fatal error: Uncaught exception 'ErrorException' with message 'Wrong parameter count for
Stack trace:
#0 [internal function]: exception_error_handler(2, 'Wrong parameter...', '/home/bjori/php
#1 /home/bjori/php/cleandocs/test.php(8): strpos()
#2 {main}
  thrown in /home/bjori/tmp/ex.php on line 8
```

93.3.1 ErrorException::__construct

```
public ErrorException::__construct() ([ string $message = "" [, int $code = 0 [, int
    $severity = 1 [, string $filename = __FILE__ [, int $lineno = __LINE__ [, Exception
    $previous = NULL ]]]]]] )
```

- message 抛出的异常消息内容。
- code 异常代码。
- severity 异常的严重级别。
- filename 抛出异常所在的文件名。

- lineno 抛出异常所在的行号。
- previous 异常链中的前一个异常。

93.3.2 ErrorException::getSeverity

ErrorException::getSeverity —获取并返回异常的严重程度，此函数没有参数。

```
final public int ErrorException::getSeverity ( void )
```

Example 113 ErrorException::getSeverity() 例子

```
<?php
try {
    throw new ErrorException("Exception message", 0, 75);
} catch(ErrorException $e) {
    echo "This exception severity is: " . $e->getSeverity();
}
?>
```

以上例程的输出类似于:

```
This exception severity is: 75
```

Part XIII

Generator

Chapter 94

Overview

生成器提供了一种更容易的方法来实现简单的对象迭代，而且相比定义类实现 Iterator 接口的方式，性能开销和复杂性大大降低。

生成器允许用户在 foreach 代码块中编写代码来迭代一组数据而不需要在内存中创建一个数组，否则会使内存达到上限，或者占据可观的处理时间。相反，生成器允许写一个生成器函数，就像一个普通的自定义函数一样，不过和普通函数只返回一次不同的是生成器可以根据需要 yield 多次来生成需要迭代的值。

一个简单的例子就是使用生成器来重新实现 range() 函数。

以前标准的 range() 函数需要在内存中生成一个数组包含每一个在它范围内的值，然后返回该数组，结果就是会产生多个很大的数组，例如调用 range(0, 1000000) 将导致内存占用超过 100 MB。

现在做为一种替代方法，通过生成器可以实现一个 xrange() 生成器，只需要足够的内存来创建 Iterator 对象并在内部跟踪生成器的当前状态，结果就是只需要不到 1K 字节的内存。

Example 114　使用生成器实现 range()

```
<?php
function xrange($start, $limit, $step = 1) {
    if ($start < $limit) {
        if ($step <= 0) {
            throw new LogicException('Step must be +ve');
        }
        for ($i = $start; $i <= $limit; $i += $step) {
            yield $i;
        }
    } else {
        if ($step >= 0) {
            throw new LogicException('Step must be -ve');
```

```
        }
        for ($i = $start; $i >= $limit; $i += $step) {
            yield $i;
        }
    }
}

echo 'Single digit odd numbers from range(): ';
foreach (range(1, 9, 2) as $number) {
    echo "$number ";
}
echo "\n";

echo 'Single digit odd numbers from xrange(): ';
foreach (xrange(1, 9, 2) as $number) {
    echo "$number ";
}
```

当第一次调用一个生成器函数时会返回一个内部生成器类的对象，这个对象实现了 Iterator 接口。

生成器类的对象一般是作为一个向前迭代的对象，并且提供可以调用的方法来修改生成器的状态（包括向其发送值或从其中获取返回值）。

和 Iterator 对象相比，生成器的主要优势在于简单，这样相比实现一个 Iterator 类就可以使用更少的代码，而且生成器代码可读性更好。例如，下面的生成器函数和 Iterator 类是等价的。

```
<?php
function getLinesFromFile($fileName) {
    if (!$fileHandle = fopen($fileName, 'r')) {
        return;
    }

    while (false !== $line = fgets($fileHandle)) {
        yield $line;
    }

    fclose($fileHandle);
}

// versus...
```

```
class LineIterator implements Iterator {
    protected $fileHandle;

    protected $line;
    protected $i;

    public function __construct($fileName) {
        if (!$this->fileHandle = fopen($fileName, 'r')) {
            throw new RuntimeException('Couldn\'t open file "' . $fileName . '"');
        }
    }

    public function rewind() {
        fseek($this->fileHandle, 0);
        $this->line = fgets($this->fileHandle);
        $this->i = 0;
    }

    public function valid() {
        return false !== $this->line;
    }

    public function current() {
        return $this->line;
    }

    public function key() {
        return $this->i;
    }

    public function next() {
        if (false !== $this->line) {
            $this->line = fgets($this->fileHandle);
            $this->i++;
        }
    }

    public function __destruct() {
        fclose($this->fileHandle);
    }
}
```

生成器提供的灵活性是有代价的，即生成器仅支持前向迭代，并且迭代一旦开始就无法倒回，这就意味着同一个生成器不能被迭代多次，而且生成器除非通过调用生成器函数来重新生成，或者通过 clone 关键字进行复制。

94.1 Syntax

一个生成器函数看起来就是一个普通的函数，二者的区别如下：

- 一个普通的函数往往只返回一个值；
- 一个生成器可以 yield 生成许多个它所需要的值。

当一个生成器被调用的时候，它返回的是一个可以被遍历的对象，这样当遍历这个对象时（例如通过 foreach 循环），PHP 将会在每次需要值的时候调用生成器函数，并在产生一个值之后保存生成器的状态，于是生成器就可以在需要产生下一个值的时候恢复调用状态。

一旦不再需要产生更多的值，生成器函数可以简单退出，但是调用生成器的代码还可以继续执行，就像一个数组已经被遍历完了。

注意，一个生成器不可以返回值，否则会产生一个编译错误，不过 return 空是一个有效的语法并且它将会终止生成器继续执行。

94.2 Yield

生成器函数的核心是 yield 关键字。

生成器函数的最简单的调用形式看起来像一个 return 申明，不同之处在于普通 return 会返回值并终止函数的执行，而 yield 会返回一个值给循环调用此生成器的代码并且只是暂停执行生成器函数。

```
function gen_one_to_three() {
    for ($i = 1; $i <= 3; $i++) {
        //注意变量$i的值在不同的yield之间是保持传递的。
        yield $i;
    }
}

$generator = gen_one_to_three();
foreach ($generator as $value) {
    echo "$value\n";
}
?>
```

上述示例的输出如下：

```
1
2
3
```

生成器对象在内部会为生成的值配备连续的整型索引（就像一个非关联的数组）。

如果在一个表达式上下文（例如在一个赋值表达式的右侧）中使用 yield，必须使用圆括号把 yield 申明包围起来。

Example 115　在表达式上下文中使用 yield

```
$data = (yield $value);
```

如果没有括号就是不合法的，而且会产生一个编译错误（PHP7 除外），例如：

Example 116　不合法的 yield 用法

```
$data = yield $value;
```

使用把圆括号把 yield 声明括起来的语法可以和生成器对象的 Generator::send() 方法配合使用。

94.3 Array

生成器同样支持 PHP 的关联键值对数组，所以除了生成简单的值，也可以使用生成器在生成值的时候指定键名。

在下面的示例中可以看到，生成一个键值对与定义一个关联数组十分相似。

Example 117　生成一个键值对

```
<?php
/*
 * 下面每一行是用分号分割的字段组合，第一个字段将被用作键名。
 */

$input = <<<'EOF'
1;PHP;Likes dollar signs
2;Python;Likes whitespace
3;Ruby;Likes blocks
EOF;

function input_parser($input) {
    foreach (explode("\n", $input) as $line) {
```

```
        $fields = explode(';', $line);
        $id = array_shift($fields);

        yield $id => $fields;
    }
}

foreach (input_parser($input) as $id => $fields) {
    echo "$id:\n";
    echo "   $fields[0]\n";
    echo "   $fields[1]\n";
}
```

上述示例的输出如下：

```
1:
    PHP
    Likes dollar signs
2:
    Python
    Likes whitespace
3:
    Ruby
    Likes blocks
```

和之前生成简单值类型一样，在一个表达式上下文中生成键值对也需要使用圆括号进行包围。

```
$data = (yield $key => $value);
```

94.4 NULL

Yield 可以在没有参数传入的情况下被调用来生成一个 NULL 值并配对一个自动的键名。

Example 118 生成 NULLs

```
<?php
function gen_three_nulls() {
    foreach (range(1, 3) as $i) {
        yield;
    }
}
```

```
var_dump(iterator_to_array(gen_three_nulls()));
?>
```

上述示例的结果如下：

```
array(3) {
  [0]=>
  NULL
  [1]=>
  NULL
  [2]=>
  NULL
}
```

除了生成 NULL 值之外，生成函数可以像使用值一样来使用引用生成，这个和从函数返回一个引用一样，也就是通过在函数名前面加一个引用符号。

Example 119 使用引用来生成值

```
<?php
function &gen_reference() {
    $value = 3;

    while ($value > 0) {
        yield $value;
    }
}

/*
 * 可以在循环中修改$number的值，而生成器是使用的引用值来生成，
 * 所以gen_reference()内部的$value值也会跟着变化。
 */
foreach (gen_reference() as &$number) {
    echo (--$number).'... ';
}
```

以上示例会输出：

```
2... 1... 0...
```

94.5 From

yield from 可以用来实现生成器委托（generator delegation），或者说从其他的生成器、Traversable 对象或数组中通过 yield from 关键字来执行遍历取值。

外部生成器可以从内部生成器中、对象或数组中遍历取值直到它不再有效，然后在外部生成器中继续执行。

如果生成器被用于 yield from，那么 yield from 表达式将会返回内部生成器返回的任何值。

```
<?php
function count_to_ten() {
    yield 1;
    yield 2;
    yield from [3, 4];
    yield from new ArrayIterator([5, 6]);
    yield from seven_eight();
    yield 9;
    yield 10;
}

function seven_eight() {
    yield 7;
    yield from eight();
}

function eight() {
    yield 8;
}

foreach (count_to_ten() as $num) {
    echo "$num ";
}
```

上述示例的输出如下：

```
1 2 3 4 5 6 7 8 9 10
```

Example 120 使用 yield from 获取返回值

```
<?php
function count_to_ten() {
    yield 1;
    yield 2;
```

```
    yield from [3, 4];
    yield from new ArrayIterator([5, 6]);
    yield from seven_eight();
    return yield from nine_ten();
}

function seven_eight() {
    yield 7;
    yield from eight();
}

function eight() {
    yield 8;
}

function nine_ten() {
    yield 9;
    return 10;
}

$gen = count_to_ten();
foreach ($gen as $num) {
    echo "$num ";
}
echo $gen->getReturn();
?>
```

上述示例的输出如下：

```
1 2 3 4 5 6 7 8 9 10
```

Part XIV

Interface

Chapter 95

Overview

Chapter 96

Traversable

Traversable 接口检测类是否支持使用 foreach 进行遍历。

抽象的基本接口不能单独实现，必须由 IteratorAggregate 接口或 Iterator 接口实现，不过实现了 Traversable 接口的内部类可以使用 foreach 进行遍历，无需实现 IteratorAggregate 接口或 Iterator 接口。

实际上，Traversable 接口是一个内部引擎接口，无法在脚本中实现，必须使用 IteratorAggregate 接口或 Iterator 接口代替。

在实现扩展 Traversable 接口的接口时，需要确保 implements 子句中的名称之前已经列出了 IteratorAggregate 接口或 Iterator 接口。

96.1 Interface synopsis

```
Traversable {
}
```

Traverable 接口没有提供方法，其作用是作为所有实现 Traversable 接口的类（可遍历类）的基础接口。

Chapter 97

Iterator Interface

Iterator 接口是用于外部迭代器或对象的接口，可以在内部自行迭代。

97.1 Interface synopsis

```
Iterator extends Traversable {
/* Methods */
abstract public mixed current ( void )
abstract public scalar key ( void )
abstract public void next ( void )
abstract public void rewind ( void )
abstract public boolean valid ( void )
}
```

PHP 已经通过 SPL 为许多日常任务提供了对应的迭代器。

下面的示例说明了使用 foreach 和迭代器时调用方法的顺序。

```
<?php

/**
 * Created by PhpStorm.
 * User: zgy
 * Date: 12/7/16
 * Time: 9:16 PM
 */
class myIterator implements Iterator {
   private $position = 0;
   private $array = array(
```

```
        "firstelement",
        "secondelement",
        "lastelement"
    );

    public function __construct() {
        $this->position = 0;
    }

    function rewind() {
        var_dump(__METHOD__);
        $this->position = 0;
    }

    function current() {
        var_dump(__METHOD__);
        return $this->array[$this->position];
    }

    function key() {
        var_dump(__METHOD__);
        return $this->position;
    }

    function next() {
        var_dump(__METHOD__);
        ++$this->position;
    }

    function valid() {
        var_dump(__METHOD__);
        return isset($this->array[$this->position]);
    }
}

$it = new myIterator();
foreach ($it as $key => $value) {
    var_dump($key, $value);
    echo "\n";
}
```

上述示例输出如下：

```
string(18) "myIterator::rewind"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(0)
string(12) "firstelement"

string(16) "myIterator::next"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(1)
string(13) "secondelement"

string(16) "myIterator::next"
string(17) "myIterator::valid"
string(19) "myIterator::current"
string(15) "myIterator::key"
int(2)
string(11) "lastelement"

string(16) "myIterator::next"
string(17) "myIterator::valid"
```

97.2 Iterator::current()

```
abstract public mixed Iterator::current ( void )
```

返回当前元素（可以是任何类型）

97.3 Iterator::key()

```
abstract public scalar Iterator::key ( void )
```

返回当前元素的 key（成功时返回标量，失败返回 NULL，发生故障时报告 E_NOTICE）。

97.4 Iterator::next()

```
abstract public void Iterator::next ( void )
```

把当前位置移动到下一个元素（在每一次 foreach 循环后都会调用该方法），并且忽略任何返回值。

97.5 Iterator::rewind()

```
abstract public void Iterator::rewind ( void )
```

返回到 Iterator 的第一个元素，Iterator::rewind() 是 foreach 循环开始后调用的第一个方法，不会在 foreach 循环后执行，而且会忽略任何返回值。

97.6 Iterator::valid()

```
abstract public boolean Iterator::valid ( void )
```

检查当前位置是否合法，在 Iterator::rewind() 或 Iterator::next() 方法执行后都会调用该方法。

返回值会被强制转换为布尔值，然后再进行计算，成功返回 true，失败返回 false。

Chapter 98

IteratorAggregate Interface

IteratorAggregate 接口用于创建一个外部迭代器。

98.1 Interface synopsis

```
IteratorAggregate extends Traversable {
/* Methods */
abstract public Traversable getIterator ( void )
}
```

```
<?php
class myData implements IteratorAggregate {
    public $property1 = "Public property one";
    public $property2 = "Public property two";
    public $property3 = "Public property three";

    public function __construct() {
        $this->property4 = "last property";
    }

    public function getIterator() {
        return new ArrayIterator($this);
    }
}

$obj = new myData;

foreach($obj as $key => $value) {
```

```
    var_dump($key, $value);
    echo "\n";
}
```

上述示例的输出如下:

```
string(9) "property1"
string(19) "Public property one"

string(9) "property2"
string(19) "Public property two"

string(9) "property3"
string(21) "Public property three"

string(9) "property4"
string(13) "last property"
```

98.2 IteratorAggregate::getIterator()

```
abstract public Traversable IteratorAggregate::getIterator ( void )
```

返回一个外部迭代器（实现了 Iterator 或 Traversable 的对象的实例），失败时会抛出 Exception 异常。

Chapter 99

Throwable Interface

99.1 Interface synopsis

```
Throwable {
/* Methods */
abstract public string getMessage ( void )
abstract public int getCode ( void )
abstract public string getFile ( void )
abstract public int getLine ( void )
abstract public array getTrace ( void )
abstract public string getTraceAsString ( void )
abstract public Throwable getPrevious ( void )
abstract public string __toString ( void )
}
```

Throwable 接口是任何可以抛出异常/错误（包括 Error 和 Exception）的对象的基础接口。PHP 类不能直接实现 Throwable 接口，必须通过扩展 Exception 接口实现。

99.2 Throwable::getMessage()

```
abstract public string Throwable::getMessage ( void )
```

返回与抛出对象相关联的消息。

99.3 Throwable::getCode()

```
abstract public int Throwable::getCode ( void )
```

返回与抛出对象相关联的错误代码。

通常情况下，在 Exception 中的异常代码为整数，但是也可以是 Exception 子类中的其他类型（例如作为 PDOException 中的字符串）。

99.4 Throwable::getFile()

```
abstract public string Throwable::getFile ( void )
```

返回与抛出对象相关联的文件名。

99.5 Throwable::getLine()

```
abstract public int Throwable::getLine ( void )
```

返回与抛出对象被实例化的行号。

99.6 Throwable::getTrace()

```
abstract public array Throwable::getTrace ( void )
```

以与 debug_backtrace() 相同的格式返回堆栈跟踪数组。

99.7 Throwable::getTraceAsString()

```
abstract public string Throwable::getTraceAsString ( void )
```

以字符串形式返回堆栈跟踪。

99.8 Throwable::getPrevious()

```
abstract public Throwable Throwable::getPrevious ( void )
```

返回任何先前的 Throwable 对象（例如作为第三个参数传递给给 Exception::__construct()），否则返回 NULL。

99.9 Throwable::__toString()

```
abstract public string Throwable::__toString ( void )
```

返回抛出对象的字符串表示形式。

Chapter 100

ArrayAccess Interface

提供把访问对象作为数组的接口。

100.1 Interface synopsis

```
ArrayAccess {
/* Methods */
abstract public boolean offsetExists ( mixed $offset )
abstract public mixed offsetGet ( mixed $offset )
abstract public void offsetSet ( mixed $offset , mixed $value )
abstract public void offsetUnset ( mixed $offset )
}
```

```
<?php
class obj implements ArrayAccess {
    private $container = array();

    public function __construct() {
        $this->container = array(
            "one"   => 1,
            "two"   => 2,
            "three" => 3,
        );
    }

    public function offsetSet($offset, $value) {
        if (is_null($offset)) {
            $this->container[] = $value;
```

```
        } else {
            $this->container[$offset] = $value;
        }
    }

    public function offsetExists($offset) {
        return isset($this->container[$offset]);
    }

    public function offsetUnset($offset) {
        unset($this->container[$offset]);
    }

    public function offsetGet($offset) {
        return isset($this->container[$offset]) ? $this->container[$offset] : null;
    }
}

$obj = new obj;

var_dump(isset($obj["two"]));
var_dump($obj["two"]);
unset($obj["two"]);
var_dump(isset($obj["two"]));
$obj["two"] = "A value";
var_dump($obj["two"]);
$obj[] = 'Append 1';
$obj[] = 'Append 2';
$obj[] = 'Append 3';
print_r($obj);
```

上述示例的输出如下:

```
bool(true)
int(2)
bool(false)
string(7) "A value"
obj Object
(
    [container:obj:private] => Array
        (
            [one] => 1
```

```
            [three] => 3
            [two] => A value
            [0] => Append 1
            [1] => Append 2
            [2] => Append 3
        )
)
```

100.2 ArrayAccess::offsetExists()

```
abstract public boolean ArrayAccess::offsetExists ( mixed $offset )
```

检测偏移是否存在，实现 ArrayAccess 接口的对象在调用 isset() 或 empty() 时就会执行 ArrayAccess::offsetExists()。

实际上，在调用 empty() 时，如果 ArrayAccess::offsetExists() 返回 true 时，ArrayAccess::offsetGet() 将被调用并检查是否为空。

$offset 是需要检查的偏移。

实现 ArrayAccess 接口的对象在执行 ArrayAccess::offsetExists() 时，成功返回 TRUE，失败返回 FALSE，如果返回值不是布尔值则会被强制转换为布尔型。

```
<?php
class obj implements arrayaccess {
    public function offsetSet($offset, $value) {
        var_dump(__METHOD__);
    }
    public function offsetExists($var) {
        var_dump(__METHOD__);
        if ($var == "foobar") {
            return true;
        }
        return false;
    }
    public function offsetUnset($var) {
        var_dump(__METHOD__);
    }
    public function offsetGet($var) {
        var_dump(__METHOD__);
        return "value";
    }
}
```

```
$obj = new obj;

echo "Runs obj::offsetExists()\n";
var_dump(isset($obj["foobar"]));

echo "\nRuns obj::offsetExists() and obj::offsetGet()\n";
var_dump(empty($obj["foobar"]));

echo "\nRuns obj::offsetExists(), *not* obj:offsetGet() as there is nothing to get\n";
var_dump(empty($obj["foobaz"]));
```

上述示例将会返回：

```
Runs obj::offsetExists()
string(17) "obj::offsetExists"
bool(true)

Runs obj::offsetExists() and obj::offsetGet()
string(17) "obj::offsetExists"
string(14) "obj::offsetGet"
bool(false)

Runs obj::offsetExists(), *not* obj:offsetGet() as there is nothing to get
string(17) "obj::offsetExists"
bool(true)
```

100.3 ArrayAccess::offsetGet()

```
abstract public mixed ArrayAccess::offsetGet ( mixed $offset )
```

返回指定偏移处的值，并且在使用 empty() 检查偏移是否存在时会自动调用该方法。

$offset 是要检索的偏移量。

ArrayAccess::offsetGet() 可以返回任何类型的值。

PHP 已经放宽了原型检查而且可以通过引用返回，从而使得对于 ArrayAccess 对象的重载数组的间接修改成为可能。

- 直接修改是完全替换数组维度的值。例如，$obj[6]=7。
- 间接修改仅仅更改维度的一部分，或者尝试通过引用来分配维度。例如，$obj[6][7]=7 或者 $var=&$obj[6]。

递增（++）和递减（--）操作同样也是使用间接修改的方式实现。

- 直接修改会触发 ArrayAccess::offsetGet()；
- 间接修改会触发 ArrayAccess::offsetSet()。

ArrayAccess::offsetGet() 的实现必须能够通过引用返回，否则会报告 E_NOTICE 消息。

100.4 ArrayAccess::offsetSet()

```
abstract public void ArrayAccess::offsetSet ( mixed $offset , mixed $value )
```

为指定的偏移分配一个值（该方法没有返回值）。

如果指定的值不可用，那么 offset 参数将被设置为 NULL，例如：

```
<?php
$arrayaccess[] = "first value";
$arrayaccess[] = "second value";
print_r($arrayaccess);
```

上述示例的结果如下：

```
Array
(
    [0] => first value
    [1] => second value
)
```

注意，在引用赋值（assignment by reference），否则使用 ArrayAccess 来进行数组维度的间接修改。

从某种意义上说，它们并不是直接更改维度，而是通过引用其他变量来实现更改子维度或子属性或分配数组维度。

只有 ArrayAccess::offsetGet() 通过引用返回时，操作才会成功。

100.5 ArrayAccess::offsetUnset()

```
abstract public void ArrayAccess::offsetUnset ( mixed $offset )
```

取消（删除）偏移，而且执行 ArrayAccess::unset() 来进行类型转换时不会调用该方法。

Chapter 101

Serializable Interface

Serializable 接口用于自定义序列化。

实现 Serializable 接口的类不再支持 __sleep() 和 __wakeup() 魔术方法。

实现 Serializable 接口后，只要对象实例需要序列化就会执行 Serializable 接口中的 Serializable::serialize() 方法，不会调用 __destruct() 方法或其他任何有副作用的方法（除非在方法内部进行了自定义）。

如果已知类的序列化数据进行反序列化，那么就会调用 unserialize() 方法而不是 __construct() 方法，如果用户需要执行标准的构造函数，那么需要自己在 Serializable::unserialize() 方法自己指定。

注意，实现 Serialize 接口的类的一个旧实例（在类实现 Serialize 接口之前已经被序列化）进行反序列化时，还是会调用 __wakeup() 而不是 Serializable::serialize() 方法，因此可以用来进行有目的的代码迁移。

101.1 Serialize synopsis

```
Serializable {
/* Methods */
abstract public string serialize ( void )
abstract public void unserialize ( string $serialized )
}
```

```
<?php
class obj implements Serializable {
   private $data;
   public function __construct() {
       $this->data = "My private data";
```

```
    }
    public function serialize() {
        return serialize($this->data);
    }
    public function unserialize($data) {
        $this->data = unserialize($data);
    }
    public function getData() {
        return $this->data;
    }
}

$obj = new obj;
$ser = serialize($obj);

var_dump($ser);

$newobj = unserialize($ser);

var_dump($newobj->getData());
```

上述示例的输出如下:

```
string(38) "C:3:"obj":23:{s:15:"My private data";}"
string(15) "My private data"
```

101.2 Serialize::serialize()

```
abstract public string Serializable::serialize ( void )
```

返回对象的序列化（字符串）表示（或 NULL）。

Serializable::serialize() 方法替代了对象的析构函数，在 Serializable::serialize() 方法之后不会再调用 __destruct() 方法。

如果序列化结果不是字符串或 NULL，Serializable::serialize() 方法会抛出 Exception 异常。

101.3 Serialize::unserialize()

```
abstract public void Serializable::unserialize ( string $serialized )
```

对对象进行反序列化，没有返回值。

Serializable::unserialize() 方法替代了对象的构造函数，在 Serializable::unserialize() 方法之后不会再调用 __construct() 方法。

Chapter 102

Closure Class

102.1 Closure synopsis

Closure 接口用于表示匿名函数的类。

PHP 支持依赖 Closure 接口来实现闭包函数和 yield，而且 PHP 允许在匿名函数创建以后进一步控制匿名函数类。

```
Closure {
/* Methods */
private __construct ( void )
public static Closure bind ( Closure $closure , object $newthis [, mixed $newscope = "
    static" ] )
public Closure bindTo ( object $newthis [, mixed $newscope = "static" ] )
public mixed call ( object $newthis [, mixed $... ] )
}
```

除了 Closure::__construct()、Closure::bind()、Closure::bindTo() 和 Closure::call() 方法之外，匿名类还有一个 Closure::__invoke() 方法来保持和其他实现魔术方法调用的类的一致性，不过在这里，Closure::__invoke() 方法不再用于魔术方法。

102.2 Closure::__construct()

```
private Closure::__construct ( void )
```

匿名类的对象以匿名函数指定的形式被实例化。

匿名类不允许实例化也没有返回值，强制调用会返回 E_RECOVERABLE_ERROR 错误。

102.3 Closure::bind()

```
public static Closure Closure::bind ( Closure $closure , object $newthis [, mixed
    $newscope = "static" ] )
```

使用指定的绑定对象和类作用域来复制闭包，可以把 Closure::bind() 方法看作是 Closure::bindTo() 方法的静态版本，成功返回新的闭包对象，失败返回 FALSE。

- closure
  The anonymous functions to bind.
- newthis
  The object to which the given anonymous function should be bound, or NULL for the closure to be unbound.
- newscope
  The class scope to which associate the closure is to be associated, or 'static' to keep the current one. If an object is given, the type of the object will be used instead. This determines the visibility of protected and private methods of the bound object. It is not allowed to pass (an object of) an internal class as this parameter.

```
<?php
class A {
   private static $sfoo = 1;
   private $ifoo = 2;
}
$cl1 = static function() {
   return A::$sfoo;
};
$cl2 = function() {
   return $this->ifoo;
};

$bcl1 = Closure::bind($cl1, null, 'A');
$bcl2 = Closure::bind($cl2, new A(), 'A');
echo $bcl1(), "\n";
echo $bcl2(), "\n";
```

上述示例的输出如下：

```
1
2
```

102.4 Closure::bindTo()

```
public Closure Closure::bindTo ( object $newthis [, mixed $newscope = "static" ] )
```

创建并返回一个新的匿名函数（即闭包对象），该函数具有与原匿名函数一样的主体和绑定变量，不过可能具有不同的绑定对象和新的类作用域，失败返回 FALSE。

The “bound object” determines the value $this will have in the function body and the “class scope” represents a class which determines which private and protected members the anonymous function will be able to access. Namely, the members that will be visible are the same as if the anonymous function were a method of the class given as value of the newscope parameter.

Static closures cannot have any bound object (the value of the parameter newthis should be NULL), but this function can nevertheless be used to change their class scope.

This function will ensure that for a non-static closure, having a bound instance will imply being scoped and vice-versa. To this end, non-static closures that are given a scope but a NULL instance are made static and non-static non-scoped closures that are given a non-null instance are scoped to an unspecified class.

如果只是需要复制匿名函数，那么可以改用 clone（克隆）来代替 Closure::bindTo()。

- newthis

  The object to which the given anonymous function should be bound, or NULL for the closure to be unbound.

- newscope

  The class scope to which associate the closure is to be associated, or 'static' to keep the current one. If an object is given, the type of the object will be used instead. This determines the visibility of protected and private methods of the bound object.

```
<?php
class A {
   function __construct($val) {
      $this->val = $val;
   }
   function getClosure() {
      //returns closure bound to this object and scope
      return function() { return $this->val; };
   }
}

$ob1 = new A(1);
$ob2 = new A(2);
```

```
$cl = $ob1->getClosure();
echo $cl(), "\n";
$cl = $cl->bindTo($ob2);
echo $cl(), "\n";
```

上述示例的结果如下：

```
1
2
```

102.5 Closure::call()

```
public mixed Closure::call ( object $newthis [, mixed $... ] )
```

绑定和调用闭包（Closure::call() 会把闭包临时绑定到 $newthis，并且使用任何给定的参数来调用它），并且返回闭包的返回值。

- $newthis

  The object to bind the closure to for the duration of the call.

- $...

  Zero or more parameters, which will be given as parameters to the closure.

```
<?php
class Value {
   protected $value;

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

   public function getValue() {
      return $this->value;
   }
}

$three = new Value(3);
$four = new Value(4);

$closure = function ($delta) { var_dump($this->getValue() + $delta); };
$closure->call($three, 4);
$closure->call($four, 4);
```

上述示例的结果如下：

```
int(7)
int(8)
```

Chapter 103

Generator Class

103.1 Generator sysopsis

```
Generator implements Iterator {
/* Methods */
public mixed current ( void )
public mixed getReturn ( void )
public mixed key ( void )
public void next ( void )
public void rewind ( void )
public mixed send ( mixed $value )
public mixed throw ( Exception $exception )
public bool valid ( void )
public void __wakeup ( void )
}
```

从生成器类可以返回生成器对象，生成器对象不能通过 new 操作实例化。

103.2 Generator::current()

```
public mixed Generator::current ( void )
```

获取 yield 操作的值

103.3 Generator::getReturn()

```
public mixed Generator::getReturn ( void )
```

获取生成器当前执行（也就是说生成器要执行一次）的返回值。

```
<?php
$gen = (function() {
    yield 1;
    yield 2;

    return 3;
})();

foreach ($gen as $val) {
    echo $val, PHP_EOL;
}

echo $gen->getReturn(), PHP_EOL;
```

上述示例的结果如下：

```
1
2
3
```

103.4 Generator::key()

```
public mixed Generator::key ( void )
```

返回 yield 操作的 key

```
<?php
function Gen()
{
    yield 'key' => 'value';
}

$gen = Gen();

echo "{$gen->key()} => {$gen->current()}";
```

上述操作的结果如下：

```
key => value
```

103.5 Generator::next()

```
public void Generator::next ( void )
```

恢复生成器的执行，没有返回值。

103.6 Generator::rewind()

```
public void Generator::rewind ( void )
```

回滚生成器，没有返回值，不过如果此时迭代已经开始，那么会抛出异常。

103.7 Generator::send()

```
public mixed Generator::send ( mixed $value )
```

把上一次 yield 操作的结果发送给生成器，并恢复生成器的执行，然后返回本次 yield 操作的值。

如果在调用 Generator::send() 方法时，生成器没有 yield 表达式，就会先把生成器回滚到第一个 yield 表达式，因此实际上没有必要使用 Generator::next() 来启动生成器（类似于 Python）。

- $value

  Value to send into the generator. This value will be the return value of the yield expression the generator is currently at.

```
<?php
function printer() {
    echo "I'm printer!".PHP_EOL;
    while (true) {
        $string = yield;
        echo $string.PHP_EOL;
    }
}

$printer = printer();
$printer->send('Hello world!');
$printer->send('Bye world!');
```

上述示例的结果如下：

```
I'm printer!
Hello world!
Bye world!
```

103.8 Generator::throw()

```
public mixed Generator::throw ( Exception $exception )
```

把异常抛出到生成器中，并恢复生成器的执行，并返回本次 yield 操作的结果。

Generator::throw() 的行为和使用 `throw $exception` 语句替换当前 yield 表达式的结果相同。

如果在调用 Generator::throw() 方法时生成器已经关闭，异常将被抛出在调用者的上下文中。

```
<?php
function gen() {
    echo "Foo\n";
    try {
        yield;
    } catch (Exception $e) {
        echo "Exception: {$e->getMessage()}\n";
    }
    echo "Bar\n";
}

$gen = gen();
$gen->rewind();
$gen->throw(new Exception('Test'));
```

上述示例的结果如下：

```
Foo
Exception: Test
Bar
```

103.9 Generator::valid()

```
public bool Generator::valid ( void )
```

检查生成器是否已关闭，如果生成器已关闭则返回 FALSE，如果生成器未关闭则返回 TRUE。

103.10 Generator::__wakeup()

```
public void Generator::__wakeup ( void )
```

没有返回值，只有在生成器无法被序列化时会抛出异常。

Part XV

Function Handling

Chapter 104

Overview

函数处理模块提供的函数处理与处理函数有关的各种操作。

104.1 Requirement

函数处理模块不需要其他扩展支持。

104.2 Resource Type

函数处理模块没有定义资源类型。

104.3 Build-in Module

函数处理模块是 PHP 核心的一部分，由 PHP 核心直接支持。

104.4 Runtime Configure

函数处理模块没有定义常量。

104.5 Predefined Constants

PHP 为文件系统函数族预先没有定义常量

Chapter 105

Functions

105.1 call_user_func_array()

调用回调函数，并把一个数组参数作为回调函数的参数

105.2 call_user_func()

把第一个参数作为回调函数调用

105.3 create_function()

创建匿名（lambda 样式）函数

105.4 forward_static_call_array()

调用静态方法并将参数作为数组传递

105.5 forward_static_call()

调用一个静态方法

105.6 func_get_arg()

返回参数列表的某一项

105.7 func_get_args()

返回一个包含函数参数列表的数组

105.8 func_num_args()

返回传递给函数的参数个数

105.9 function_exists()

如果给定的函数已经被定义就返回 TRUE

105.10 get_defined_functions()

返回所有已定义函数的数组

105.11 register_shutdown_function()

注册在 PHP 进程结束时执行的函数

105.12 register_tick_function()

在每个 tick 上注册一个执行的函数

105.13 unregister_tick_function()

在每个 tick 上取消注册的一个执行的函数

Part XVI

Context

Chapter 106

Overview

PHP 提供了可以用于所有的文件系统或数据流封装协议的上下文参数和选项，其中上下文（context）由 stream_context_create() 创建。

- 选项可以通过 stream_context_set_option() 进行设置；
- 参数可以通过 stream_context_set_params() 进行设置。

106.1 Socket context

套接字上下文选项可用于所有工作在套接字上的封装协议（例如 TCP、HTTP 和 FTP 等）。

106.1.1 bindto

bindto 设置用户 PHP 访问网络的指定的 IP 地址（IPv4 或 IPv6 其中的一个）和/或端口号（例如 `ip:port`）。如果把 IP 或端口号设置为 0，则允许系统自动选择 IP 或端口号。

- 把 bindto 设置为 0:0 来强制使用 IPv4 协议
- 把 bindto 设置为 [0]0 来强制使用 IPv6 协议

在正常操作环境中，FTP 需要打开两条 Socket 连接（控制连接和数据连接），因此无法使用 bindto 来指定端口号。

106.1.2 backlog

backlog 设置用于限制套接字侦听队列中未完成的连接数，仅对 stream_socket_server() 有效。

```
<?php
// connect to the internet using the '192.168.0.100' IP
```

```
$opts = array(
    'socket' => array(
        'bindto' => '192.168.0.100:0',
    ),
);

// connect to the internet using the '192.168.0.100' IP and port '7000'
$opts = array(
    'socket' => array(
        'bindto' => '192.168.0.100:7000',
    ),
);

// connect to the internet using port '7000'
$opts = array(
    'socket' => array(
        'bindto' => '0:7000',
    ),
);

// create the context...
$context = stream_context_create($opts);

// ...and use it to fetch the data
echo file_get_contents('http://www.example.com', false, $context);
```

106.2 HTTP context

HTTP 上下文选项可以设置提供给 http:// 和 https:// 传输协议的 context 选项。

106.2.1 method

string 类型 - 远程服务器支持的 GET，POST 或其它 HTTP 方法，默认值是 GET。

106.2.2 header

string 类型 - 请求期间发送的额外 header，使用 header 选项可以覆盖其他值（例如 User-agent:，Host: 和 Authentication:）。

106.2.3 user_agent

string 类型 - 要发送的 header User-Agent: 的值。如果在 header context 选项中没有指定 user-agent，此值将被使用（默认使用 php.ini 中设置的 user_agent）。

106.2.4 content

string 类型 - 在 header 后面要发送的额外数据，通常使用 POST 或 PUT 请求。

106.2.5 proxy

string 类型 - URI 指定的代理服务器的地址（例如 `tcp://proxy.example.com:5100`）。

106.2.6 request_fulluri

boolean 类型 - 当设置为 TRUE 时（默认值是 FALSE），在构建请求时将使用整个 URI 。例如：`GET http://www.example.com/path/to/file.html HTTP/1.0`。

虽然上述是一个非标准的请求格式，但是某些代理服务器需要它。

106.2.7 follow_location

integer 类型 - 跟随 Location header 的重定向。设置为 0 以禁用（默认值为 1）。

106.2.8 max_redirects

integer 类型 - 跟随重定向的最大次数（默认值是 20）。值为 1 或更少则意味不跟随重定向。

106.2.9 protocol_version

float 类型 - HTTP 协议版本（默认值是 1.0）。如果设置为 1.1，可能需要自己处理分块传输解码操作。

106.2.10 timeout

float 类型 - 读取超时时间，单位为秒（s），用 float 指定（例如 10.5），默认会使用 php.ini 中设置的 default_socket_timeout。

106.2.11 ignore_errors

boolean 类型 - 即使是故障状态码依然获取内容，默认值为 FALSE。

Example 121 获取一个页面并发送 POST 数据

```
<?php

$postdata = http_build_query(
    array(
        'var1' => 'some content',
        'var2' => 'doh'
    )
);

$opts = array('http' =>
    array(
        'method' => 'POST',
        'header' => 'Content-type: application/x-www-form-urlencoded',
        'content' => $postdata
    )
);

$context = stream_context_create($opts);

$result = file_get_contents('http://example.com/submit.php', false, $context
    );
```

Example 122 忽略重定向并获取 header 和内容

```
<?php

$url = "http://www.example.org/header.php";

$opts = array('http' =>
    array(
        'method' => 'GET',
        'max_redirects' => '0',
        'ignore_errors' => '1'
    )
);

$context = stream_context_create($opts);
```

```php
$stream = fopen($url, 'r', false, $context);

// header information as well as meta data
// about the stream
var_dump(stream_get_meta_data($stream));

// actual data at $url
var_dump(stream_get_contents($stream));
fclose($stream);
```

底层的 Socket 数据流上下文选项可以通过 tcp://和或 ssl://来为上层的 http://或 https://提供服务。

如果需要实现可以跟随 HTTP 状态码来进行重定向的流封装器，stream_get_meta_data() 返回的 wrapper_data 不一定包含实际应用于索引 0 处的内容数据的 HTTP 状态行。

```
array (
  'wrapper_data' =>
  array (
    0 => 'HTTP/1.0 301 Moved Permantenly',
    1 => 'Cache-Control: no-cache',
    2 => 'Connection: close',
    3 => 'Location: http://example.com/foo.jpg',
    4 => 'HTTP/1.1 200 OK',
    ...
```

如果第一个请求返回 301 重定向，那么流包装器需要自动跟随重定向来获得 200 响应 (也就是上述的索引 4)。

106.3 FTP context

FTP 上下文选项支持 ftp://和 ftps://数据传输。

106.3.1 overwrite

boolean 类型 - 是否允许覆盖远程服务器上已有的文件，仅适用于写入模式（上传），默认值为 FALSE。

106.3.2 resume_pos

integer 类型 - 开始传输的文件偏移量，仅适用于读取模式（下载），默认值为 0（即文件开头）。

106.3.3 proxy

string 类型 - 通过 HTTP 代理服务器（例如 `tcp://squid.example.com:8000`）发送代理 FTP 请求，仅适用于文件读取操作。

底层的 Socket 数据流上下文选项可以通过 tcp://和或 ssl://来为上层的 ftp://或 ftps://提供服务。

106.4 SSL context

SSL 上下文选项提供 ssl://和 tls://传输协议的上下文选项清单。

106.4.1 peer_name

string 类型 - 要连接的服务器名称。如果未设置，那么服务器名称将根据打开 SSL 流的主机名称猜测得出。

106.4.2 verify_peer

boolean 类型 - 是否需要验证 SSL 证书，默认为 FALSE。

106.4.3 verify_peer_name

boolean 类型 - 需要验证对等名称。

106.4.4 allow_self_signed

boolean 类型 - 是否允许自签名证书（默认为 FALSE）。需要配合 verify_peer 参数使用。

当 verify_peer 参数为 true 时才会根据 allow_self_signed 参数值来决定是否允许自签名证书。

106.4.5 cafile

string 类型 - 当设置 verify_peer 为 true 时用来验证远端证书所用到的 CA 证书，cafile 的值为 CA 证书在本地文件系统的全路径及文件名。

106.4.6 capath

capath 类型 - 如果未设置 cafile，或者 cafile 所指的文件不存在时，就会在 capath 所指定的目录搜索适用的证书。

capath 指定的目录必须是已经经过哈希处理的证书目录，这里的已经经过哈希处理的证书目录（hashed certificate）是指使用类似 c_rehash 命令将目录中的.pem 和.crt 文件扫描并提取哈希码，然后根据此哈希码创建文件链接，可以便于快速查找证书。

106.4.7 local_cert

string 类型 - 本地证书路径，要求必须是 PEM 格式，并且包含本地的证书及私钥，或者也可以包含证书颁发者证书链，也可以通过 local_pk 指定包含私钥的独立文件。

106.4.8 local_pk

string 类型 - 如果使用独立的文件来存储证书（local_cert）和私钥，那么使用此选项来指明私钥文件的路径。

106.4.9 passphrase

string 类型 - local_cert 文件的密码。

106.4.10 CN_match

string 类型 - 期望的远端证书的 CN 名称，PHP 会进行有限的通配符匹配，如果服务器给出的 CN 名称和本地访问的名称不匹配，则视为连接失败。

CN_match 选项已被废弃，并被替换为 peer_name 选项。

106.4.11 verify_depth

integer 类型 - 如果证书链条层次太深，超过了本选项的设定值，则终止验证。默认情况下不限制证书链条层次深度。

106.4.12 ciphers

string 类型 - 设置可用的密码列表，默认值为 DEFAULT。

106.4.13 capture_peer_cert

boolean 类型 - 如果设置为 TRUE 将会在上下文中创建 peer_certificate 选项，该选项中包含远端证书。

106.4.14 capture_peer_cert_chain

boolean 类型 - 如果设置为 TRUE 将会在上下文中创建 peer_certificate_chain 选项，该选项中包含远端证书链条。

106.4.15 SNI_enabled

boolean 类型 - 设置为 TRUE 将启用服务器名称指示（server name indication）。启用 SNI 将允许同一 IP 地址使用多个证书。

106.4.16 SNI_server_name

string 类型 - 如果设置此参数，那么其设置值将被视为 SNI 服务器名称。如果未设置，那么服务器名称将基于打开 SSL 流的主机名称猜测得出。

SNI_server_name 选项已被废弃，并替换为 peer_name。

106.4.17 disable_compression

boolean 类型 - 如果设置，则禁用 TLS 压缩，有助于减轻恶意攻击。

106.4.18 peer_fingerprint

当远程服务器证书的摘要和指定的散列值不相同的时候，终止操作。

- 当使用 string 时，会根据字符串的长度来检测所使用的散列算法：“md5”（32 字节）还是“sha1”（40 字节）。
- 当使用 array 时，数组的键表示散列算法名称，其对应的值是预期的摘要值。

ssl://是 https://和 ftps://的底层传输协议，因此 ssl://的上下文选项同样适用于 https://和 ftps://的上下文。

PHP 必须配合 OpenSSL 编译才能支持 SNI，同时也支持使用 OPENSSL_TLSEXT_SERVER_NAME 来探测 SNI 服务器名称。

106.5 CURL context

CURL 上下文选项在 CURL 扩展被编译（通过`--with-curlwrappers` 选项）时可用。

106.5.1 method

string 类型 - GET、POST 或者其他远程服务器支持的 HTTP 方法，默认为 GET。

106.5.2 header

string 类型 - 额外的请求标头，而且这个值会覆盖通过其他选项设定的值（例如 User-agent:，Host:，，Authentication:）。

106.5.3 user_agent

string 类型 - 设置请求时 User-Agent 标头的值，默认为 php.ini 中的 user_agent 设定值。

106.5.4 content

string 类型 - 在头部之后发送的额外数据，该选项在 GET 和 HEAD 请求中不使用。

106.5.5 proxy

string 类型 - 用于指定代理服务器的 URI 地址（例如 `tcp://proxy.example.com:5100`）。

106.5.6 max_redirects

integer 类型 - 最大重定向次数，如果为 1 或者更小则代表不会跟随重定向，默认为 20。

106.5.7 curl_verify_ssl_host

boolean 类型 - 是否校验服务器，默认为 FALSE。
curl_verify_ssl_host 选项在 HTTP 和 FTP 协议中均可使用。

106.5.8 curl_verify_ssl_peer

boolean 类型 - 是否要求对使用的 SSL 证书进行校验，默认为 FALSE。
curl_verify_ssl_peer 选项在 HTTP 和 FTP 协议中均可使用。

Example 123 获取一个页面，并以 POST 发送数据

```
<?php
$postdata = http_build_query(
    array(
        'var1' => 'some content',
        'var2' => 'doh'
    )
);

$opts = array('http' =>
```

```
    array(
        'method' => 'POST',
        'header' => 'Content-type: application/x-www-form-urlencoded',
        'content' => $postdata
    )
);

$context = stream_context_create($opts);
$result = file_get_contents('http://example.com/submit.php', false, $context
    );
```

106.6 Phar context

phar:// 封装（wrapper）的上下文（context）选项。

106.6.1 compress

int 类型 - Phar compression constants 中的一个。

- Phar::None
- Phar::COMPRESSED
- Phar::GZ
- Phar::BZ2

106.6.2 metadata

mixed 类型 - Phar 元数据（metadata）。

```
<?php
// make sure it doesn't exist
@unlink('brandnewphar.phar');
try {
    $p = new Phar(dirname(__FILE__) . '/brandnewphar.phar', 0, 'brandnewphar.phar');
    $p['file.php'] = '<?php echo "hello"';
    $p->setMetadata(array('bootstrap' => 'file.php'));
    var_dump($p->getMetadata());
} catch (Exception $e) {
    echo 'Could not create and/or modify phar:', $e;
}
```

106.7 MongoDB context

mongodb://传输协议的上下文选项。

106.7.1 log_cmd_insert

callable 类型 - 插入文档时调用的回调函数

106.7.2 log_cmd_delete

callable 类型 - 删除文档时调用的回调函数

106.7.3 log_cmd_update

callable 类型 - 更新文档时调用的回调函数

106.7.4 log_cmd_batch

callable 类型 - 在执行写入批处理时调用的回调函数

106.7.5 log_reply

callable 类型 - 从 MongoDB 读取回复时调用的回调函数

106.7.6 log_getmore

callable 类型 - 从 MongoDB 游标检索更多结果时调用的回调函数

106.7.7 log_killcursor

callable 类型 - 执行 killcursor 时调用的回调函数

106.8 Context parameters

Context 参数（parameters）可以设置为由函数 stream_context_set_params() 返回的 context。

106.8.1 notification

callable 类型 - 当一个流（stream）上发生事件时，notification 指定的回调函数将被调用。

Part XVII

Protocol

Chapter 107

Overview

PHP 提供了很多内置 URL 风格的封装协议，可用于类似 fopen()、copy()、file_exists() 和 filesize() 的文件系统函数。除了这些封装协议，还能通过 stream_wrapper_register() 来注册自定义的封装协议。

注意，用于描述一个封装协议的 URL 语法仅支持 scheme://的语法，scheme:/ 和 scheme: 语法是不支持的。

107.1 file://

文件系统协议是 PHP 使用的默认封装协议，file://协议用于访问本地文件系统资源。

当指定了一个相对路径（不以/、\、\\或 Windows 盘符开头的路径）提供的路径时，PHP 将基于当前的工作目录，在很多情况下是脚本所在的目录（除非被修改了）。

使用 PHP CLI 时，目录默认是脚本被调用时所在的目录。

- /path/to/file.ext
- relative/path/to/file.ext
- fileInCwd.ext
- C:/path/to/winfile.ext
- C:\path\to\winfile.ext
- \\smbserver\share\path\to\winfile.ext
- file:///path/to/file.ext

另外，在某些函数（例如 fopen() 和 file_get_contents()）中，include_path 会被可选地搜索，也作为相对的路径。

表 107.1: file://封装协议概要

属性	支持
受 allow_url_fopen 影响	No
允许读取	Yes
允许写入	Yes
允许添加	Yes
允许同时读和写	Yes
支持 stat()	Yes
支持 unlink()	Yes
支持 rename()	Yes
支持 mkdir()	Yes
支持 rmdir()	Yes

107.2 http://

http://协议允许通过 HTTP 1.0 的 GET 方法以只读访问文件或资源，并且在启用 openssl 扩展后可以使用 https://协议。

HTTP 连接是只读的，不支持对一个 HTTP 资源进行写数据或者复制文件（比如发送 POST 和 PUT 请求），可以在 HTTP Contexts 的支持下实现 POST 请求和 PUT 请求等。

- http://example.com
- http://example.com/file.php?var1=val1&var2=val2
- http://user:password@example.com
- https://example.com
- https://example.com/file.php?var1=val1&var2=val2
- https://user:password@example.com

HTTP 请求会附带一个 Host: 头来兼容基于域名的虚拟主机。如果在 php.ini 文件中或字节流上下文（context）配置了 user_agent 字符串，那么它也会被包含在请求之中。

数据流允许读取资源的 body，而 headers 则储存在了 $http_response_header 变量里。

stream_get_meta_data() 可以从封装协议文件指针中取得报头/元数据，因此如果需要知道文档资源来自哪个 URL（经过所有重定向的处理后），需要使用 stream_get_meta_data() 处理数据流返回的系列响应报头（response headers）。

如果设置了 From: 头且没有被上下文（Context）选项和参数覆盖，那么 From: 头可以被 from 指令使用。

表 107.2: http://封装协议概要

属性	支持
受 allow_url_fopen 限制	Yes
允许读取	Yes
允许写入	No
允许添加	No
允许同时读和写	N/A
支持 stat()	No
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

Example 124　检测重定向后最终的 URL

```
<?php
$url = 'http://www.example.com/redirecting_page.php';

$fp = fopen($url, 'r');

$meta_data = stream_get_meta_data($fp);
foreach ($meta_data['wrapper_data'] as $response) {

    /* 我们是否被重定向了？ */
    if (strtolower(substr($response, 0, 10)) == 'location: ') {

        /* 更新我们被重定向后的 $url */
        $url = substr($response, 10);
    }
}
```

107.3 ftp://

ftp://协议用于访问 FTP(s) 资源并允许通过 FTP 读取存在的文件、创建新文件以及追加（append）文件，在启用 openssl 扩展后可以支持 ftps://。

如果服务器不支持被动（passive）模式的 FTP，连接会失败。

- ftp://example.com/pub/file.txt
- ftp://user:password@example.com/pub/file.txt
- ftps://example.com/pub/file.txt
- ftps://user:password@example.com/pub/file.txt

ftp://协议在打开文件后既可以执行读操作也可以执行写操作，不过二者不能同时进行。

- 如果远程文件已经存在于 FTP 服务器上，如果尝试打开并写入文件时未指定上下文（context）选项 overwrite，连接会失败。
- 如果要通过 FTP 覆盖存在的文件，需要指定上下文（context）的 overwrite 选项来打开和写入，或者可以使用 FTP 扩展来代替 ftp://协议操作。

如果服务器不支持 SSL，这个连接会降级（falls back）到普通未加密的 ftp。

如果设置了 php.ini 中的 from 指令，这个值会被作为匿名 FTP（anonymous）的密码。

表 107.3: ftp://封装协议概要

属性	支持
受 allow_url_fopen 影响	Yes
允许读取	Yes
允许写入	Yes (新文件/启用 overwrite 后已存在的文件)
允许添加	Yes
允许同时读和写	No
支持 stat()	仅 filesize()、filetype()、file_exists()、is_file()、is_dir() 和 filemtime()
支持 unlink()	Yes
支持 rename()	Yes
支持 mkdir()	Yes
支持 rmdir()	Yes

107.4 php://

php://协议用于访问输入/输出流（I/O stream）、标准输入输出和错误描述符、内存和磁盘中的临时文件流以及可以操作其他读取/写入文件资源的过滤器。

PHP 提供了常量 STDIN、STDOUT 和 STDERR 来代替手工打开 php://stdin、php://stdout 和 php://stderr 封装器。

- php://stdin 是只读的；
- php://stdout 是只写的；
- php://stderr 是只写的。

107.4.1 php://fd

php://fd 允许直接访问指定的文件描述符。例如，php://fd/3 引用了文件描述符 3。

107.4.2 php://stdin

php://stdin 允许直接访问 PHP 进程相应的输入流，数据流引用了复制的文件描述符，所以如果在打开 php://stdin 并在之后关了它，那么仅仅是关闭了复制品，真正被引用的 STDIN 并不受影响。

107.4.3 php://stdout

php://stdin 允许直接访问 PHP 进程相应的输出流

107.4.4 php://stderr

php://stdin 允许直接访问 PHP 进程相应的错误流

107.4.5 php://filter

php://filter 是一种设计用于数据流打开时的筛选过滤应用的元封装器，因此对于一体式（all-in-one）的文件函数非常有用，类似 readfile()、file() 和 file_get_contents() 等函数在数据流内容读取之前没有机会应用其他过滤器。

php://filter 目标是使用以下的参数作为它路径的一部分，复合过滤链能够在一个路径上指定。

表 107.4: php://filter 参数

参数	描述
resource=< 要过滤的数据流 >	必选，指定要筛选过滤的数据流。
read=< 读链的筛选列表 >	可选，可以设定一个或多个过滤器名称，以管道符（|）分隔。
write=< 写链的筛选列表 >	可选，可以设定一个或多个过滤器名称，以管道符（|）分隔。
<; 两个链的筛选列表 >	任何没有以 read= 或 write= 作前缀的筛选器列表会视情况应用于读或写链。

Example 125 php://filter/resource=< 待过滤的数据流 >

```
<?php
/* 这简单等同于:
  readfile("http://www.example.com");
  实际上没有指定过滤器 */

readfile("php://filter/resource=http://www.example.com");
```

`resource=` 参数必须位于 php://filter 的末尾，并且指向需要过滤筛选的数据流。

Example 126 php://filter/read=< 读链需要应用的过滤器列表 >

```
<?php
/* 这会以大写字母输出 www.example.com 的全部内容 */
readfile("php://filter/read=string.toupper/resource=http://www.example.com")
    ;

/* 这会和以上所做的一样，但还会用 ROT13 加密。 */
readfile("php://filter/read=string.toupper|string.rot13/resource=http://www.
    example.com");
```

`read=` 参数采用一个或以管道符 | 分隔的多个过滤器名称。

Example 127 php://filter/write=< 写链需要应用的过滤器列表 >

```
<?php
/* 这会通过 rot13 过滤器筛选出字符 "Hello World"
```

```
 然后写入当前目录下的 example.txt */
file_put_contents("php://filter/write=string.rot13/resource=example.txt","
    Hello World");
```

`write=` 参数采用一个或以管道符 | 分隔的多个过滤器名称。

107.4.6 php://input

php://input 是一个可以访问请求的原始数据的只读流，可以反复使用。

在处理 POST 请求时，最好使用 php://input 来代替 $HTTP_RAW_POST_DATA，不依赖于特定的 php.ini 指令，而且 $HTTP_RAW_POST_DATA 在这样的情况下默认没有填充，因此比激活 always_populate_raw_post_data 潜在需要更少的内存。

如果传入的数据类型是 `enctype="multipart/form-data"` 时，php://input 是无效的。

PHP 在保存 POST 请求体数据时可以打开另一个 php://input 数据流并重新读取，这种情况只是针对 POST 请求而非其他请求方式（比如 PUT 或者 PROPFIND）。

PHP SAPI 的实现允许可以多次读取 php://input 打开的数据流，而且 php://input 数据流支持 seek 操作。

107.4.7 php://output

php://output 是一个只写的数据流，允许使用和 print 和 echo 相同的方式写入到输出缓冲区。

107.4.8 php://memory

php://memory 和 php://temp 是一个类似文件包装器的数据流，都允许读写临时数据，两者的唯一区别如下：

- php://memory 总是把数据储存在内存中；
- php://temp 会在内存量达到预定义的限制后（默认是 2MB）存入临时文件中，临时文件位置的决定和 sys_get_temp_dir() 的方式一致。

107.4.9 php://temp

php://temp 是一个允许读写临时数据的类似文件包装器的数据流，其内存限制可以通过添加 /maxmemory:NN 来控制，NN 是以字节为单位、保留在内存的最大数据量，超过则使用临时文件。

Example 128 设置 php://temp 开始使用临时文件前的最大内存限制

```
<?php
// Set the limit to 5 MB.
$fiveMBs = 5 * 1024 * 1024;
$fp = fopen("php://temp/maxmemory:$fiveMBs", 'r+');

fputs($fp, "hello\n");

// Read what we have written.
rewind($fp);
echo stream_get_contents($fp);
```

表 107.5: php://封装协议概要

属性	支持
首先于 allow_url_fopen	No
首先于 allow_url_include	仅 php://input、php://stdin、php://memory 和 php://temp。
允许读取	仅 php://stdin、php://input、php://fd、php://memory 和 php://temp。
允许写入	仅 php://stdout、php://stderr、php://output、php://fd、php://memory 和 php://temp。
允许追加	仅 php://stdout、php://stderr、php://output、php://fd、php://memory 和 php://temp (等于写入)
允许同时读写	仅 php://fd、php://memory 和 php://temp。
支持 stat()	仅 php://memory 和 php://temp。
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

属性	支持
仅仅支持 stream_select()	php://stdin、php://stdout、php://stderr、php://fd 和 php://temp。

107.5 zlib://

zlib://协议用于访问压缩流，其功能类似 gzopen()，而且其数据流还能被 fread() 和其他文件系统函数使用，不过现在建议使用 compress.zlib:// 作为替代，后来 zip 扩展又注册了 zip://协议。

- compress.zlib://file.gz
- compress.bzip2://file.bz2
- zip://archive.zip#dir/file.txt

compress.zlib://、compress.bzip2:// 和 gzopen()、bzopen() 是相等的，并且可以在不支持 fopencookie 的系统中使用。

表 107.6: zlib://封装协议概要

属性	支持
受限于 allow_url_fopen	No
允许读取	Yes
允许写入	Yes（除了 zip://）
允许附加	Yes（除了 zip://）
允许同时读写	No
支持 stat()	No（使用普通的 file:// 封装器统计压缩文件）
支持 unlink()	No（使用 file:// 封装器删除压缩文件）
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

107.6 data://

data://协议用于访问符合 RFC 2397 规范的数据流（例如 `data://text/plain;base64`）。

Example 129 打印 data:// 的内容

```
<?php
// 打印 "I love PHP"
echo file_get_contents('data://text/plain;base64,SSBsb3ZlIFBIUAo=');
```

表 107.7: data://封装协议概要

属性	支持
受限于 allow_url_fopen	No
受限于 allow_url_include	Yes
允许读取	Yes
允许写入	No
允许追加	No
允许同时读写	No
支持 stat()	No
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

Example 130 获取媒体类型

```
<?php
$fp   = fopen('data://text/plain;base64,', 'r');
$meta = stream_get_meta_data($fp);

// 打印 "text/plain"
echo $meta['mediatype'];
```

107.7 glob://

glob://协议用于查找匹配的文件路径模式。

表 107.8: glob://封装协议概要

属性	支持
受限于 allow_url_fopen	No
受限于 allow_url_include	No
允许读取	No
允许写入	No
允许附加	No
允许同时读写	No
支持 stat()	No
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

```
<?php
// 循环 ext/spl/examples/ 目录里所有 *.php 文件
// 并打印文件名和文件尺寸
$it = new DirectoryIterator("glob://ext/spl/examples/*.php");
foreach($it as $f) {
    printf("%s: %.1FK\n", $f->getFilename(), $f->getSize()/1024);
}
```

上述示例的结果可能如下：

```
tree.php: 1.0K
findregex.php: 0.6K
findfile.php: 0.7K
dba_dump.php: 0.9K
nocvsdir.php: 1.1K
phar_from_dir.php: 1.0K
ini_groups.php: 0.9K
directorytree.php: 0.9K
```

```
dba_array.php: 1.1K
class_tree.php: 1.8K
```

107.8 phar://

phar://协议用于 PHP 归档。

表 107.9: glob://封装协议概要

属性	支持
支持 allow_url_fopen	No
支持 allow_url_include	No
允许读取	Yes
允许写入	Yes
允许附加	No
允许同时读写	Yes
支持 stat()	Yes
支持 unlink()	Yes
支持 rename()	Yes
支持 mkdir()	Yes
支持 rmdir()	Yes

107.9 ssh2://

ssh2://协议用于执行 SSH 操作，ssh2://封装器默认未激活，而且必须安装 ssh2 扩展才能使用 ssh2://协议封装器。

- ssh2.shell://user:pass@example.com:22/xterm
- ssh2.exec://user:pass@example.com:22/usr/local/bin/somecmd
- ssh2.tunnel://user:pass@example.com:22/192.168.0.1:14
- ssh2.sftp://user:pass@example.com:22/path/to/filename

除了支持传统的 URI 登录信息，ssh2 封装协议也支持通过 URL 的主机（host）部分来复用打开连接。

- ssh2.shell://
- ssh2.exec://
- ssh2.tunnel://

- ssh2.sftp://
- ssh2.scp://

表 107.10: ssh2://封装协议概要

属性	ssh2.shell	ssh2.exec	ssh2.tunnel	ssh2.sftp	ssh2.scp
受 allow_url_fopen 影响	Yes	Yes	Yes	Yes	Yes
允许读取	Yes	Yes	Yes	Yes	Yes
允许写入	Yes	Yes	Yes	Yes	No
允许追加	No	No	No	Yes（需要服务器支持）	No
允许同时读和写	Yes	Yes	Yes	Yes	No
支持 stat()	No	No	No	Yes	No
支持 unlink()	No	No	No	Yes	No
支持 rename()	No	No	No	Yes	No
支持 mkdir()	No	No	No	Yes	No
支持 rmdir()	No	No	No	Yes	No

ssh2://协议支持的上下文选项如下：

- ssesion 重复使用预连接的 ssh2 资源
- sftp 重复使用预先分配的 sftp 资源
- methods 密钥交换（key exchange）、主机密钥（hostkey）、cipher、压缩和 MAC 方法
- callbacks
- username 以该用户名连接
- password 使用的密码来进行密码验证
- pubkey_file 用于验证的公钥（public key）文件
- privkey_file 用于验证的私钥（private key）文件
- env 需要设置的环境变量的关联数组
- term 在分配一个 pty 时请求的终端类型
- term_width 在分配一个 pty 时请求的终端宽度
- term_height 在分配一个 pty 时请求的终端宽度高度
- term_units erm_width 和 term_height 的单位

Example 131 从一个活动连接中打开字节流

```
<?php
```

```
$session = ssh2_connect('example.com', 22);
ssh2_auth_pubkey_file($session, 'username', '/home/username/.ssh/id_rsa.pub'
    ,
                                        '/home/username/.ssh/id_rsa', 'secret')
                                            ;
$stream = fopen("ssh2.tunnel://$session/remote.example.com:1234", 'r');
```

Example 132 通过 $session 重复使用预连接的 ssh2 资源

```
<?php
$session = ssh2_connect('example.com', 22);
ssh2_auth_pubkey_file($session, 'username', '/home/username/.ssh/id_rsa.pub'
    ,
                                        '/home/username/.ssh/id_rsa', 'secret')
                                            ;
$connection_string = "ssh2.sftp://$session/";
unset($session);
$stream = fopen($connection_string . "path/to/file", 'r');
```

unset() 可以关闭会话，而且 $connecting_string 不保持对 $session 变量的引用。

107.10 rar://

rar://协议用于访问 RAR 文件流，并且需要启用 rar 扩展才能使用 rar://包装器。

rar://协议支持使用存储在存档中的 RAR 存档（相对或绝对）的 url 编码路径，可选的 asterik(*)，可选的数字符号 (#) 和可选的 URL 编码的条目名称。

rar://协议要求指定条目名称时需要数字符号，条目名称中的前导正斜杠是可选的。例如，`rar://<url encoded archive name>[*][#[<url encoded entry name>]]`

rar://协议可以打开文件或目录，其中：

- 打开目录时，指定星号符号强制返回未编码的目录条目名称。
- 打开目录时，未指定星号返回 url encoded 数据。

rar://允许包装器正确地使用 PHP 内置的功能（例如 RecursiveDirectoryIterator），而且也可能存在类似 url 编码的文件名。

如果不包括 # 号和条目名称部分，rar://将显示归档的根目录。

不同于常规目录，rar://生成的流不包含诸如修改时间（mtime）等的信息，因为根目录不存储在存档中的单个条目中。

在使用 RecursiveDirectoryIterator 的包装器时，需要在访问根目录的时候在 URL 中包含数字符号，以便可以正确构建内部的 URL。

表 107.11: rar://封装协议概要

属性	支持
支持 allow_url_fopen	No
支持 allow_url_include	No
允许读取	Yes
允许写入	No
允许附加	No
允许同时读写	No
支持 stat()	Yes
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

rar://协议支持的上下文选项包括 open_password、file_password 和 volume_callback 等。

- open_password - 用于加密存档标题（如果有）的密码。WinRAR 将使用与头文件密码相同的密码加密所有文件，因此对于具有加密头的文件，file_password 将被忽略。
- file_password - 用于加密文件的密码（如果有）。如果文件头也被加密，则此选项将被忽略以支持 open_password。
- valume_callback - 确定丢失卷路径的回调。

Example 133 遍历 RAR 归档

```
<?php
class MyRecDirIt extends RecursiveDirectoryIterator {
    function current() {
        return rawurldecode($this->getSubPathName()) .
            (is_dir(parent::current())?" [DIR]":"");
    }
}

$f = "rar://" . rawurlencode(dirname(__FILE__)) .
    DIRECTORY_SEPARATOR . 'dirs_and_extra_headers.rar#';

$it = new RecursiveTreeIterator(new MyRecDirIt($f));

foreach ($it as $s) {
    echo $s, "\n";
```

```
}
```

Example 134 打开加密文件（标头加密）

```
<?php
$stream = fopen("rar://" .
   rawurlencode(dirname(__FILE__)) . DIRECTORY_SEPARATOR .
   'encrypted_headers.rar' . '#encfile1.txt', "r", false,
   stream_context_create(
      array(
         'rar' =>
            array(
               'open_password' => 'samplepassword'
            )
         )
      )
   );
var_dump(stream_get_contents($stream));
/* creation and last access date is opt-in in WinRAR, hence most
 * files don't have them */
var_dump(fstat($stream));
```

上述示例可能输出如下：

```
string(26) "Encrypted file 1 contents."
Array
(
   [0] => 0
   [1] => 0
   [2] => 33206
   [3] => 1
   [4] => 0
   [5] => 0
   [6] => 0
   [7] => 26
   [8] => 0
   [9] => 1259550052
   [10] => 0
   [11] => -1
   [12] => -1
   [dev] => 0
   [ino] => 0
```

```
    [mode] => 33206
    [nlink] => 1
    [uid] => 0
    [gid] => 0
    [rdev] => 0
    [size] => 26
    [atime] => 0
    [mtime] => 1259550052
    [ctime] => 0
    [blksize] => -1
    [blocks] => -1
)
```

107.11 ogg://

ogg://协议用于音频流，因此通过 ogg://包装器读取的文件使用 OGG/Vorbis 格式的压缩音频编码，而且使用 ogg://包装器写入或追加的数据格式也是压缩音频编码。

- ogg://soundfile.ogg
- ogg:///path/to/soundfile.ogg
- ogg://http://www.example.com/path/to/soundstream.ogg

如果使用 stream_get_meta_data() 来打开并读取一个 OGG/Vorbis 格式的文件时，可以获得关于数据流的详细信息（例如 vendor 标签、任何内含的 comments、channels 数字、采样率（rate）以及用 bitrate_lower、bitrate_upper、bitrate_nominal 和 bitrate_window 描述的可变比特率范围等)。

默认情况下，需要启用 OGG/Vorbis 扩展才能激活 ogg://封装器。

表 107.12: ogg://封装协议概要

属性	支持
受限于 allow_url_fopen	No
允许读取	Yes
允许写入	Yes
允许附加	Yes
允许同时读写	No
支持 stat()	No

属性	支持
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

ogg://协议的上下文选项包括 pcm_mode、rate、bitrate、channels、comments 等。

表 107.13: ogg://协议上下文选项

Name	Usage	Default	Mode
pcm_mode	读取时使用如下 PCM 编码之一: OGGVORBIS_PCM_U8、OGGVORBIS_PCM_S8、OGGVORBIS_PCM_U16_BE、OGGVORBIS_PCM_S16_BE、OGGVORBIS_PCM_U16_LE 和 OGGVORBIS_PCM_S16_LE。(8 或 16 位，签名或未签名，大或小的 endian)	OGGVORBIS_PCM_S16_LE	读取
rate	输入数据的采样率，单位为 Hz	44100	写入/附加
bitrate	若给的值为整数，则是用固定的比特率进行编码。 （16000 到 131072）若给的值为浮点数，则使用可变的比特率（质。(-1.0 到 1.0）	128000	写入/附加
channels	要编码的声道的数量，典型为 1 (单声道), 或 2 (立体声)。最高支持 16 声道	2	写入/附加
comments	编码到音轨头部的字符串数组		写入/附加

107.12 expect://

expect://协议用于处理交互式的流，由 expect:// 封装协议打开的数据流 PTY 通过提供了对进程 stdio、stdout 和 stderr 的访问。

默认情况下，需要开启 Expect 扩展才能激活 expect://封装协议。

表 107.14: expect://封装协议概要

属性	支持
受限于 allow_url_fopen	No
允许读取	Yes
允许写入	Yes
允许附加	Yes
允许同时读写	No
支持 stat()	No
支持 unlink()	No
支持 rename()	No
支持 mkdir()	No
支持 rmdir()	No

Part XVIII

String

Chapter 108

Overview

108.1 Requirement

PHP 提供了专门的函数来分别处理字符串、正则表达式和 URL。

108.2 Resource Type

字符串处理函数没有定义资源类型。

108.3 Build-in Module

字符串处理函数由 PHP 核心直接支持。

108.4 Runtime Configure

字符串处理函数没有在 php.ini 中定义配置指令。

108.5 Predefined Constants

PHP 为字符串处理函数预先定义了一些常量，并且这些常量在此扩展编译入 PHP 或在运行时动态载入时可用。

表 108.1: 用于字符串处理函数的常量

常量名	说明
CRYPT_SALT_LENGTH integer	
CRYPT_STD_DES integer	
CRYPT_EXT_DES integer	
CRYPT_MD5 integer	
CRYPT_BLOWFISH integer	
HTML_SPECIALCHARS (integer)	
HTML_ENTITIES (integer)	
ENT_COMPAT (integer)	
ENT_QUOTES (integer)	
ENT_NOQUOTES (integer)	
ENT_IGNORE (integer)	
ENT_SUBSTITUTE (integer)	
ENT_DISALLOWED (integer)	
ENT_HTML401 (integer)	
ENT_XML1 (integer)	
ENT_XHTML (integer)	
ENT_HTML5 (integer)	

常量名	说明
CHAR_MAX (integer)	
LC_CTYPE (integer)	
LC_NUMERIC (integer)	
LC_TIME (integer)	
LC_COLLATE (integer)	
LC_MONETARY (integer)	
LC_ALL (integer)	
LC_MESSAGES (integer)	
STR_PAD_LEFT (integer)	
STR_PAD_RIGHT (integer)	
STR_PAD_BOTH (integer)	

表 108.2: 用于 openlog() 设备的常量

常量名	说明
LOG_AUTH	security/authorization messages (use LOG_AUTHPRIV instead in systems where that constant is defined)
LOG_AUTHPRIV	security/authorization messages (private)
LOG_CRON	clock daemon (cron and at)
LOG_DAEMON	other system daemons
LOG_KERN	kernel messages
LOG_LOCAL0 ... LOG_LOCAL7	reserved for local use, these are not available in Windows
LOG_LPR	line printer subsystem
LOG_MAIL	mail subsystem

常量名	说明
LOG_NEWS	USENET news subsystem
LOG_SYSLOG	messages generated internally by syslogd
LOG_USER	generic user-level messages
LOG_UUCP	UUCP subsystem

表 108.3: 用于 syslog() 的优先级（按降序排列）的常量

常量名	说明
LOG_EMERG	system is unusable
LOG_ALERT	action must be taken immediately
LOG_CRIT	critical conditions
LOG_ERR	error conditions
LOG_WARNING	warning conditions
LOG_NOTICE	normal, but significant, condition
LOG_INFO	informational message
LOG_DEBUG	debug-level message

表 108.4: 用于 dns_get_record() 选项的常量

常量名	说明
DNS_A	IPv4 Address Resource
DNS_MX	Mail Exchanger Resource
DNS_CNAME	Alias (Canonical Name) Resource
DNS_NS	Authoritative Name Server Resource
DNS_PTR	Pointer Resource
DNS_HINFO	Host Info Resource
DNS_SOA	Start of Authority Resource
DNS_TXT	Text Resource
DNS_ANY	Any Resource Record. On most systems this returns all resource records, however it should not be counted upon for critical uses. Try DNS_ALL instead.

常量名	说明
DNS_AAAA	IPv6 Address Resource
DNS_ALL	Iteratively query the name server for each available record type.

Chapter 109

Functions

109.1 addcslashes()

以 C 语言风格使用反斜线转义字符串中的字符

109.2 addslashes()

使用反斜线引用字符串

109.3 bin2hex()

把包含数据的二进制字符串转换为十六进制值

109.4 chop()

rtrim 的别名

109.5 chr()

返回指定的字符

109.6 chunk_split()

将字符串分割成小块

109.7 convert_cyr_string()

将字符由一种 Cyrillic 字符转换成另一种

109.8 convert_uudecode()

解码一个 uuencode 编码的字符串

109.9 convert_uuencode()

使用 uuencode 编码一个字符串

109.10 count_chars()

返回字符串所用字符的信息

109.11 crc32()

计算一个字符串的 crc32 多项式

109.12 crypt()

单向字符串散列

109.13 echo()

输出一个或多个字符串

109.14 explode()

使用一个字符串分割另一个字符串

109.15 fprintf()

将格式化后的字符串写入到流

109.16 get_html_translation_table()

返回使用 htmlspecialchars 和 htmlentities 后的转换表

109.17 hebrev()

将逻辑顺序希伯来文（logical-Hebrew）转换为视觉顺序希伯来文（visual-Hebrew）

109.18 hebrevc()

将逻辑顺序希伯来文（logical-Hebrew）转换为视觉顺序希伯来文（visual-Hebrew），并且转换换行符

109.19 hex2bin()

转换十六进制字符串为二进制字符串

109.20 html_entity_decode()

将所有 HTML 实体转换为其对应的字符

109.21 htmlentities()

将所有对应的字符转换为 HTML 实体

109.22 htmlspacialchars_decode()

将特殊的 HTML 实体转换回普通字符

109.23 htmlspecialchars()

将特殊字符转换为 HTML 实体

109.24 implode()

将一个一维数组的值转化为字符串

109.25 join()

别名 implode

109.26 lcfirst()

使一个字符串的第一个字符小写

109.27 levenshtein()

计算两个字符串之间的编辑距离

109.28 localeconv()

获取数字格式信息

109.29 ltrim()

删除字符串开头的空白字符（或其他字符）

109.30 md5_file()

计算指定文件的 MD5 散列值

109.31 md5()

计算字符串的 MD5 散列值

109.32 metaphone()

计算字符串的 metaphone 键

109.33 money_format()

将数字格式化成货币字符串

109.34 nl_langinfo()

查询语言和区域设置信息

109.35 nl2br()

在字符串所有新行之前插入 HTML 换行标记

109.36 number_format()

以千位分隔符方式格式化一个数字

109.37 ord()

返回字符的 ASCII 码值

109.38 parse_str()

将字符串解析成多个变量

109.39 print()

输出字符串

109.40 printf()

输出格式化字符串

109.41 quoted_printable_decode()

将 quoted-printable 字符串转换为 8-bit 字符串

109.42 quoted_printable_encode()

将 8-bit 字符串转换成 quoted-printable 字符串

109.43 quotemeta()

转义元字符集

109.44 rtrim()

删除字符串末端的空白字符（或者其他字符）

109.45 setlocale()

设置地区信息

109.46 sha1_file()

计算文件的 sha1 散列值

109.47 sha1()

计算字符串的 sha1 散列值

109.48 similar_text()

计算两个字符串的相似度

109.49 soundex()

计算字符串的 soundex 键

109.50 sprintf()

返回格式化后的字符串

109.51 sscanf()

根据指定格式解析输入的字符

109.52 str_getcsv()

解析 CSV 字符串为一个数组

109.53 str_ireplace()

str_replace() 的忽略大小写版本

109.54 str_pad()

使用另一个字符串填充字符串为指定长度

109.55 str_repeat()

重复一个字符串

109.56 str_replace()

子字符串替换

109.57 str_rot13()

对字符串执行 ROT13 转换

109.58 str_shuffle()

随机打乱一个字符串

109.59 str_split()

将字符串转换为数组

109.60 str_word_count()

返回字符串中单词的使用情况

109.61 strcasecmp()

二进制安全比较字符串（不区分大小写）

109.62 strchr()

别名 strstr

109.63 strcmp()

二进制安全字符串比较

109.64 strcoll()

基于区域设置的字符串比较

109.65 strcspn()

获取不匹配遮罩的起始子字符串的长度

109.66 strip_tags()

从字符串中去除 HTML 和 PHP 标记

109.67 stripcslashes()

反引用一个使用 addcslashes 转义的字符串

109.68 stripos()

查找字符串首次出现的位置（不区分大小写）

109.69 stripslashes()

反引用一个引用字符串

109.70 stristr()

strstr 函数的忽略大小写版本

109.71 strlen()

获取字符串长度

109.72 strnatcasecmp()

使用“自然顺序”算法比较字符串（不区分大小写）

109.73 strcasecmp()

使用自然排序算法比较字符串

109.74 strncmp

二进制安全比较字符串开头的若干个字符（不区分大小写）

109.75 strpbrk()

在字符串中查找一组字符的任何一个字符

109.76 strpos()

查找字符串首次出现的位置

109.77 strrchr()

查找指定字符在字符串中的最后一次出现

109.78 strrev()

反转字符串

109.79 strripos()

计算指定字符串在目标字符串中最后一次出现的位置（不区分大小写）

109.80 strrpos()

计算指定字符串在目标字符串中最后一次出现的位置

109.81 strspn()

计算字符串中全部字符都存在于指定字符集合中的第一段子串的长度。

109.82 strstr()

查找字符串的首次出现

109.83 strok()

标记分割字符串

109.84 strtolower()

将字符串转化为小写

109.85 strtoupper()

将字符串转化为大写

109.86 strtr()

转换指定字符

109.87 substr_compare()

二进制安全比较字符串（从偏移位置比较指定长度）

109.88 substr_count()

计算字串出现的次数

109.89 substr_replace()

替换字符串的子串

109.90 substr()

返回字符串的子串

109.91 trim()

去除字符串首尾处的空白字符（或者其他字符）

109.92 ucfirst()

将字符串的首字母转换为大写

109.93 ucwords()

将字符串中每个单词的首字母转换为大写

109.94 vfprintf()

将格式化字符串写入流

109.95 vprintf()

输出格式化字符串

109.96 vsprintf()

返回格式化字符串

109.97 wordwrap()

打断字符串为指定数量的字串

Part XIX

Filesystem

Chapter 110

Overview

110.1 Requirement

默认情况下，文件系统函数不需要额外的库支持，除非需要支持 Linux 的 LFS（Large File System）文件系统，那么需要以最新的 glibc 库并指定如下参数来编译 PHP：

```
-D_LARGEFILE_SOURCE -D_FILE_OFFSET_BITS=64
```

110.2 Resource Type

文件系统函数使用流（stream）作为其资源类型。

110.3 Build-in Module

文件系统函数是 PHP 核心的一部分，由 PHP 核心直接支持。

110.4 Runtime Configure

文件系统函数的行为受到 php.ini 中的相关配置项的影响。

表 110.1: Network Configuration 选项

名字	默认	可修改范围	说明
allow_url_fopen	"1"	PHP_INI_SYSTEM	TBM

名字	默认	可修改范围	说明
user_agent	NULL	PHP_INI_ALL	可用
default_socket_timeout	"60"	PHP_INI_ALL	可用
from	""	PHP_INI_ALL	
auto_detect_line_endings	"0"	PHP_INI_ALL	可用

110.4.1 allow_url_fopen

allow_url_fopen 选项激活 URL 形式的 fopen 封装协议，使得 PHP 可以访问 URL 对象（例如文件）。

默认的封装协议提供用 ftp 和 http 协议来访问远程文件，一些扩展库（例如 zlib）可能会注册更多的封装协议。

allow_url_fopen 选项只能在 php.ini 中设置（出于安全考虑）。

110.4.2 allow_url_include

allow_url_include 选项配置是否允许使用具有以下函数的 URL 感知的 fopen 包装器：

- include
- include_once
- require
- require_once

allow_url_include 选项要求 allow_url_open 选项配置为 on。

110.4.3 user_agent

user_agent 定义 PHP 发送的 User-Agent。

110.4.4 default_socket_timeout

default_socket_timeout 定义基于 socket 的流的默认超时时间（秒）。

110.4.5 from

from 选项定义匿名 ftp 的密码（email 地址）。

110.4.6 auto_detect_line_endings

auto_detect_line_endings 默认值为 off，如果设置为 on，那么 PHP 将检查通过 fgets() 和 file() 取得的数据中的行结束符号是符合 Unix、MS-DOS 还是 Macintosh 的习惯。

auto_detect_line_endings 设置为 on 时使得 PHP 可以和 Macintosh 系统交互操作。

默认 auto_detect_line_endings 的值是 Off 的原因是在检测第一行的 EOL 习惯时会有很小的性能损失，而且在 Unix 系统下使用回车符号作为项目分隔符的用户会遇到向下不兼容的行为。

110.5 Predefined Constants

PHP 为文件系统函数族预先定义了一些常量，并且这些常量可以作为 PHP 核心的一部分或者在运行时动态载入时总是可用的。

表 110.2: 用于 openlog() 选项的常量

常量名	说明
SEEK_SET (integer)	
SEEK_CUR (integer)	
SEEK_END (integer)	
LOCK_SH (integer)	
LOCK_EX (integer)	
LOCK_UN (integer)	
LOCK_NB (integer)	
GLOB_BRACE (integer)	
GLOB_ONLYDIR (integer)	
GLOB_MARK (integer)	
GLOB_NOSORT (integer)	
GLOB_NOCHECK (integer)	
GLOB_NOESCAPE (integer)	

常量名	说明
GLOB_AVAILABLE_FLAGS (integer)	
PATHINFO_DIRNAME (integer)	
PATHINFO_BASENAME (integer)	
PATHINFO_EXTENSION (integer)	
PATHINFO_FILENAME (integer)	
FILE_USE_INCLUDE_PATH (integer)	Search for filename in include_path
FILE_NO_DEFAULT_CONTEXT (integer)	
FILE_APPEND (integer)	Append content to existing file.
FILE_IGNORE_NEW_LINES (integer)	Strip EOL characters.
FILE_SKIP_EMPTY_LINES (integer)	Skip empty lines.
FILE_BINARY (integer)	Binary mode，仅用于向前兼容
FILE_TEXT (integer)	Text mode，仅用于向前兼容
INI_SCANNER_NORMAL (integer)	Normal INI scanner mode
INI_SCANNER_RAW (integer)	Raw INI scanner mode
INI_SCANNER_TYPED (integer)	Typed INI scanner mode
FNM_NOESCAPE (integer)	Disable backslash escaping.

常量名	说明
FNM_PATHNAME (integer)	Slash in string only matches slash in the given pattern.
FNM_PERIOD (integer)	Leading period in string must be exactly matched by period in the given pattern.
FNM_CASEFOLD (integer)	Caseless match. Part of the GNU extension.

Chapter 111

Functions

111.1 basename()

返回路径中的文件名部分

111.2 chgrp()

改变文件所属的组

111.3 chmod()

改变文件模式

111.4 chown()

改变文件的所有者

111.5 clearstatcache()

清除文件状态缓存

111.6 copy()

拷贝文件

111.7 delete()

参见 unlink 或 unset

111.8 dirname()

返回路径中的目录部分

111.9 disk_free_space()

返回目录中的可用空间

111.10 disk_total_space()

返回一个目录的磁盘总大小

111.11 diskfreespace()

disk_free_space 的别名

111.12 fclose()

关闭一个已打开的文件指针

111.13 feof()

测试文件指针是否到了文件结束的位置

111.14 fflush()

将缓冲内容输出到文件

111.15 fgetc()

从文件指针中读取字符

111.16 fgetcsv()

从文件指针中读入一行并解析 CSV 字段

111.17 fgets()

从文件指针中读取一行

111.18 fgetss()

从文件指针中读取一行并过滤掉 HTML 标记

111.19 file_exists()

检查文件或目录是否存在

111.20 file_get_contents()

将整个文件读入一个字符串

111.21 file_put_contents()

将一个字符串写入文件

111.22 file()

把整个文件读入一个数组中

111.23 fileatime()

取得文件的上次访问时间

111.24 filectime()

取得文件的 inode 修改时间

111.25 filegroup()

取得文件的组

111.26 fileinode()

取得文件的 inode

111.27 filemtime()

取得文件修改时间

111.28 fileowner()

取得文件的所有者

111.29 fileperms()

取得文件的权限

111.30 filesize()

取得文件大小

111.31 filetype()

取得文件类型

111.32 flock()

轻便的咨询文件锁定

111.33 fnmatch()

用模式匹配文件名

111.34 fopen()

打开文件或者 URL

111.35 fpassthru()

输出文件指针处的所有剩余数据

111.36 fputcsv()

将行格式化为 CSV 并写入文件指针

111.37 fputs()

fwrite 的别名

111.38 fread()

读取文件（可安全用于二进制文件）

111.39 fscanf()

从文件中格式化输入

111.40 fseek()

在文件指针中定位

111.41 fstat()

通过已打开的文件指针取得文件信息

111.42 ftell()

返回文件指针读/写的位置

111.43 ftruncate()

将文件截断到给定的长度

111.44 fwrite()

写入文件（可安全用于二进制文件）

111.45 glob()

寻找与模式匹配的文件路径

111.46 is_dir()

判断给定文件名是否是一个目录

111.47 is_executable()

判断给定文件名是否可执行

111.48 is_file()

判断给定文件名是否为一个正常的文件

111.49 is_link()

判断给定文件名是否为一个符号连接

111.50 is_readable()

判断给定文件名是否可读

111.51 is_uploaded_file()

判断文件是否是通过 HTTP POST 上传的

111.52 is_writable()

判断给定的文件名是否可写

111.53 is_writeable()

is_writable 的别名

111.54 lchgrp()

修改符号链接的所有组

111.55 lchown()

修改符号链接的所有者

111.56 link()

建立一个硬连接

111.57 linkinfo()

获取一个连接的信息

111.58 lstat()

给出一个文件或符号连接的信息

111.59 mkdir()

新建目录

111.60 move_uploaded_file()

将上传的文件移动到新位置

111.61 parse_ini_file()

解析一个配置文件

111.62 parse_ini_string()

解析配置字符串

111.63 pathinfo()

返回文件路径的信息

111.64 pclose()

关闭进程文件指针

111.65 popen()

打开进程文件指针

111.66 readfile()

输出文件

111.67 readlink()

返回符号连接指向的目标

111.68 realpath_cache_get()

获取真实目录缓存的详情

111.69 realpath_cache_size()

获取真实路径缓冲区的大小

111.70 realpath()

返回规范化的绝对路径名

111.71 rename()

重命名一个文件或目录

111.72 rewind()

倒回文件指针的位置

111.73 rmdir()

删除目录

111.74 set_file_buffer()

stream_set_write_buffer 的别名

111.75 stat()

给出文件的信息

111.76 symlink()

建立符号连接

111.77 tempnam()

建立一个具有唯一文件名的文件

111.78 tmpfile()

建立一个临时文件

111.79 touch()

设定文件的访问和修改时间

111.80 umask()

改变当前的 umask

111.81 unlink()

删除文件

Part XX

Variable Handling

Chapter 112

Overview

112.1 Requirement

变量处理模块是 PHP 核心的内置的功能，不需要其他扩展。

112.2 Resource Type

变量处理模块没有定义资源类型。

112.3 Build-in Module

变量处理函数由 PHP 核心直接支持。

112.4 Runtime Configure

变量处理函数的行为受到 php.ini 中的 unserialize_callback_func 配置项的影响。

表 112.1: 变量配置选项

名字	默认	可修改范围	说明
unserialize_callback_func	NULL	PHP_INI_ALL	可用

unserialize_callback_func 配置项说明如果解串行器发现有未定义类要被实例化，将会调用 unserialize() 回调函数（用该未定义类名作为参数）。

如果指定的回调函数不存在，或者此函数没有包含/实现该未定义类，则显示警告，所

以仅在确实需要实现这样的回调函数时才设置该选项。

112.5 Predefined Constants

变量处理模块没有定义常量。

Chapter 113

Functions

113.1 boolval()

获取变量的布尔值。

113.2 debug_zval_dump()

以字符串的形式 dump 内部 zend 变量的值。

113.3 doubleval()

floatval() 的别名。

113.4 empty()

检查一个变量是否为空。

113.5 floatval()

返回变量的浮点值。

113.6 get_defined_vars()

返回由所有已定义变量所组成的数组。

113.7 get_resource_type()

返回资源（resource）类型。

113.8 gettype()

获取变量的类型。

113.9 import_request_variables()

将 GET/POST/Cookie 变量导入到全局作用域中。

113.10 intval()

返回变量的整数值。

113.11 is_array()

检测变量是否是数组。

113.12 is_bool

检测变量是否是布尔型。

113.13 is_callable()

检测变量是否是合法的可调用结构。

113.14 is_double()

is_float() 的别名。

113.15 is_float()

检测变量是否是浮点型。

113.16 is_int()

检测变量是否是整型。

113.17 is_integer()

is_int() 的别名。

113.18 is_long()

is_int() 的别名。

113.19 is_null()

检测变量是否是 NULL。

113.20 is_numeric()

检测变量是否是数字或数字字符串。

113.21 is_object()

检测变量是否是对象。

113.22 is_real()

is_float() 的别名。

113.23 is_resource()

检测变量是否是资源类型。

113.24 is_scalar()

检测变量是否是标量。

113.25 is_string()

检测变量是否是字符串。

113.26 isset()

检测变量是否已被设置。

113.27 print_r()

打印关于变量的已于理解的信息。

113.28 serialize()

序列化变量来产生一个可存储的值的表示。

113.29 settype()

设置变量的类型。

113.30 strval()

获取变量的字符串值。

113.31 unserialize()

从已存储的表示中反序列化来创建 PHP 的值。

113.32 unset()

释放（删除）给定的变量。

113.33 var_dump()

打印变量的相关信息。

113.34 var_export()

输出或返回一个变量的字符串表示。

Part XXI

Date/Time

Chapter 114

Overview

114.1 MySQL

PHP 在向 MySQL 插入时间时，除了可以使用 MySQL 的 NOW() 函数之外，还可以使用 `date('Y-m-d H:i:s')`，这两种方式都是 MySQL 的 datetime 类型字段可以理解的日期/时间格式。

114.2 DateTime

DateTime 类的作用是在读、写、比较或者计算日期和时间时提供帮助。除了 DateTime 类之外，PHP 还有很多与日期和时间相关的函数，不过 DateTime 类为大多数常规使用提供了优秀的面向对象接口，而且 DateTime 类还可以处理时区。

在使用 DateTime 之前，通过 createFromFormat() 工厂方法将原始的日期与时间字符串转换为对象或使用 new DateTime 来取得当前的日期和时间。使用 format() 将 DateTime 转换回字符串用于输出。

114.3 createFromFormat()

114.4 format()

```
<?php
$raw = '22. 11. 1968';
$start = DateTime::createFromFormat('d. m. Y', $raw);

echo 'Start date: ' . $start->format('Y-m-d') ."\n";
```

对 DateTime 进行计算时可以使用 DateInterval 类。DateTime 类具有例如 add() 和 sub() 等将 DateInterval 当作参数的方法。

编写代码时注意不要认为每一天都是由相同的秒数构成的，不论是夏令时（DST）还是时区转换，使用时间戳计算都会遇到问题，应当选择日期间隔。

使用 diff() 方法来计算日期之间的间隔，它会返回新的 DateInterval，而且非常容易进行展示。

114.5 add()

114.6 sub()

114.7 diff()

```
<?php
// create a copy of $start and add one month and 6 days
$end = clone $start;
$end->add(new DateInterval('P1M6D'));

$diff = $end->diff($start);
echo 'Difference: ' . $diff->format('%m month, %d days (total: %a days)') . "\n";
// Difference: 1 month, 6 days (total: 37 days)
```

DateTime 对象之间可以直接进行比较：

```
<?php
if ($start < $end) {
    echo "Start is before end!\n";
}
```

最后一个例子来演示 DatePeriod 类。它用来对循环的事件进行迭代。向它传入开始时间、结束时间和间隔区间，会得到这其中所有的时间。

```
<?php
// output all thursdays between $start and $end
$periodInterval = DateInterval::createFromDateString('first thursday');
$periodIterator = new DatePeriod($start, $periodInterval, $end, DatePeriod::
    EXCLUDE_START_DATE);
foreach ($periodIterator as $date) {
    // output each date in the period
    echo $date->format('Y-m-d') . ' ';
}
```

Part XXII

Network

Chapter 115

Overview

PHP 核心提供的网络函数族包含一系列联网函数，而且它们的实际效率比使用 CURL 更高。

115.1 Requirement

checkdnsrr()、getmxrr() 和 dns_get_record() 函数需要 Linux 系统中安装有 Bind 来提供支持。

115.2 Resource Type

PHP 为网络函数族中的 fsockopen() 和 pfsockopen() 定义了一个文件指针资源类型。

115.3 Build-in Module

联网函数由 PHP 核心直接支持。

115.4 Runtime Configure

联网函数的行为受到 php.ini 中的 define_syslog_variables 配置项的影响。

表 115.1: Network Configuration 选项

名字	默认	可修改范围	说明
define_syslog_variables	"0"	PHP_INI_ALL	已移除

define_syslog_variables 配置项说明是否定义各种 syslog 变量(例如 $LOG_PID 和 $LOG_CRON 等)，关闭或移除 define_syslog_variables 配置项可以提供性能，推荐用户在运行时通过调用 define_syslog_variables() 定义这些变量。

115.5 Predefined Constants

PHP 为网络函数族预先定义了一些常量，并且这些常量作为 PHP 核心的一部分总是可用的。

表 115.2: 用于 openlog() 选项的常量

常量名	说明
LOG_CONS	if there is an error while sending data to the system logger, write directly to the system console
LOG_NDELAY	open the connection to the logger immediately
LOG_ODELAY	(default) delay opening the connection until the first message is logged
LOG_NOWAIT	
LOG_PERROR	print log message also to standard error
LOG_PID	include PID with each message

表 115.3: 用于 openlog() 设备的常量

常量名	说明
LOG_AUTH	security/authorization messages (use LOG_AUTHPRIV instead in systems where that constant is defined)
LOG_AUTHPRIV	security/authorization messages (private)
LOG_CRON	clock daemon (cron and at)
LOG_DAEMON	other system daemons
LOG_KERN	kernel messages

常量名	说明
LOG_LOCAL0 ... LOG_LOCAL7	reserved for local use, these are not available in Windows
LOG_LPR	line printer subsystem
LOG_MAIL	mail subsystem
LOG_NEWS	USENET news subsystem
LOG_SYSLOG	messages generated internally by syslogd
LOG_USER	generic user-level messages
LOG_UUCP	UUCP subsystem

表 115.4: 用于 syslog() 的优先级（按降序排列）的常量

常量名	说明
LOG_EMERG	system is unusable
LOG_ALERT	action must be taken immediately
LOG_CRIT	critical conditions
LOG_ERR	error conditions
LOG_WARNING	warning conditions
LOG_NOTICE	normal, but significant, condition
LOG_INFO	informational message
LOG_DEBUG	debug-level message

表 115.5: 用于 dns_get_record() 选项的常量

常量名	说明
DNS_A	IPv4 Address Resource
DNS_MX	Mail Exchanger Resource
DNS_CNAME	Alias (Canonical Name) Resource
DNS_NS	Authoritative Name Server Resource
DNS_PTR	Pointer Resource
DNS_HINFO	Host Info Resource
DNS_SOA	Start of Authority Resource

常量名	说明
DNS_TXT	Text Resource
DNS_ANY	Any Resource Record. On most systems this returns all resource records, however it should not be counted upon for critical uses. Try DNS_ALL instead.
DNS_AAAA	IPv6 Address Resource
DNS_ALL	Iteratively query the name server for each available record type.

Chapter 116

Functions

PHP 核心提供的网络函数族在实际应用中比 cURL 效率高的原因在于，前端服务器的端口回收很慢，并发高的时候有明显不足。例如，三次握手多次就会很慢，HTTP 每次请求后拿到结果都会马上关闭，如果使用 keepalive 很难完善后续操作，客户端/服务端通讯很难自定义通信协议。

116.1 checkdnsrr()

给指定的主机（域名）或者 IP 地址做 DNS 通信检查。

116.2 closelog()

关闭系统日志链接。

116.3 define_syslog_variables()

初始化所有 syslog 相关的变量。

116.4 dns_check_record()

别名 checkdnsrr()。

116.5 dns_get_mx()

别名 getmxrr()。

116.6 dns_get_record()

获取指定主机的 DNS 记录。

116.7 fsockopen()

打开一个网络连接或者一个 UNIX 套接字连接。

116.8 gethostbyaddr()

获取指定的 IP 地址对应的主机名。

116.9 gethostbyname()

获取与给定 Internet 主机名对应的 IPv4 地址。

116.10 gethostbynamel()

获取与给定 Internet 主机名对应的 IPv4 地址的列表。

116.11 gethostname()

获取主机名。

116.12 getmxrr()

获取与给定 Internet 主机名对应的 MX 记录。

116.13 getprotobyname()

获取与协议名称相关的协议号。

116.14 getprotobynumber()

获取和协议号相关的协议名。

116.15 getservbyname()

获取与 Internet 服务和协议相关联的端口号

116.16 getservbyport()

获取对应端口和协议的 Internet 服务。

116.17 header_register_callback()

注册将在 PHP 开始发送输出时调用的回调函数。

116.18 header_remove()

删除以前使用 header() 设置的旧的 HTTP 标头。

116.19 header()

发送原生 HTTP 头。

116.20 headers_list()

返回发送（或准备发送）的响应头的列表。

116.21 headers_sent()

检查是否已经发送或在何处发送了标头。

116.22 http_response_code()

获取或设置 HTTP 响应代码

116.23 inet_ntop()

将打包的互联网地址转换为人类可读的表示形式。

116.24 inet_pton()

将人类可读的 IP 地址转换为其打包的 in_addr 表示形式

116.25 ip2long()

将一个 IPV4 的字符串互联网协议转换成数字格式

116.26 long2ip()

将长整型地址转换为 IPv4 格式的 Internet 标准点分隔格式的字符串

116.27 openlog()

打开与系统日志记录器的连接

116.28 pfsockopen()

打开一个持久的网络连接或者 Unix 套接字连接。

116.29 setcookie()

发送 cookie。

116.30 setrawcookie()

发送没有对 cookie 值进行编码的 cookie

116.31 socket_get_status()

别名 stream_get_meta_data()

116.32 socket_set_blocking()

别名 stream_set_blocking()

116.33 socket_set_timeout()

别名 stream_set_timeout()

116.34 syslog()

生成一条系统日志消息。

Part XXIII

URL

Chapter 117

Overview

PHP 的 URL 函数可以处理 URL 字符串的编码、解码和解析。

117.1 Requirement

默认情况下，URL 函数不需要额外的库支持，它们是 PHP 核心的一部分。

117.2 Resource Type

URL 函数没有定义资源类型。

117.3 Build-in Module

文件系统函数是 PHP 核心的一部分，由 PHP 核心直接支持。

117.4 Runtime Configure

URL 函数没有在 php.ini 中定义配置指令。

117.4.1 allow_url_fopen

allow_url_fopen 选项激活 URL 形式的 fopen 封装协议，使得 PHP 可以访问 URL 对象（例如文件）。

默认的封装协议提供用 ftp 和 http 协议来访问远程文件，一些扩展库（例如 zlib）可能会注册更多的封装协议。

allow_url_fopen 选项只能在 php.ini 中设置（出于安全考虑)。

117.4.2 allow_url_include

allow_url_include 选项配置是否允许使用具有以下函数的 URL 感知的 fopen 包装器:

- include
- include_once
- require
- require_once

allow_url_include 选项要求 allow_url_open 选项配置为 on。

117.4.3 user_agent

user_agent 定义 PHP 发送的 User-Agent。

117.4.4 default_socket_timeout

default_socket_timeout 定义基于 socket 的流的默认超时时间（秒)。

117.4.5 from

from 选项定义匿名 ftp 的密码（email 地址)。

117.4.6 auto_detect_line_endings

auto_detect_line_endings 默认值为 off，如果设置为 on，那么 PHP 将检查通过 fgets() 和 file() 取得的数据中的行结束符号是符合 Unix、MS-DOS 还是 Macintosh 的习惯。

auto_detect_line_endings 设置为 on 时使得 PHP 可以和 Macintosh 系统交互操作。

默认 auto_detect_line_endings 的值是 Off 的原因是在检测第一行的 EOL 习惯时会有很小的性能损失，而且在 Unix 系统下使用回车符号作为项目分隔符的用户会遇到向下不兼容的行为。

117.5 Predefined Constants

PHP 为 URL 函数族预先定义了一些常量，并且这些常量可以作为 PHP 核心的一部分或者在运行时动态载入时总是可用的。

表 117.1: 用于 parse_url() 的常量

常量名	说明
PHP_URL_SCHEME (integer)	
PHP_URL_HOST (integer)	
PHP_URL_PORT (integer)	
PHP_URL_USER (integer)	
PHP_URL_PASS (integer)	
PHP_URL_PATH (integer)	
PHP_URL_QUERY (integer)	
PHP_URL_FRAGMENT (integer)	
PHP_QUERY_RFC1738 (integer)	
PHP_QUERY_RFC3986 (integer)	

Chapter 118

Functions

118.1 base64_decode()

对使用 MIME base64 编码的数据进行解码

118.2 base64_encode()

使用 MIME base64 对数据进行编码

118.3 get_headers()

取得服务器响应一个 HTTP 请求所发送的所有标头

118.4 get_meta_tags()

从一个文件中提取所有的 meta 标签 content 属性，返回一个数组

118.5 http_build_query()

生成 URL-encode 之后的请求字符串

118.6 parse_url()

解析 URL，返回其组成部分

118.7 rawurldecode()

对已编码的 URL 字符串进行解码

118.8 rawurlencode()

按照 RFC 3986 对 URL 进行编码

118.9 urldecode()

解码已编码的 URL 字符串

118.10 urlencode()

编码 URL 字符串

Part XXIV

XML

Chapter 119

Overview

XML（Extensible Markup Language，可扩展标记语言）是一种标记语言。

标记本身是指计算机所能理解的信息符号，通过此种标记，计算机之间可以处理包含各种信息的文章等。

如何定义这些标记，既可以选择国际通用的标记语言（比如 HTML），也可以使用像 XML 这样由相关人士自由决定的标记语言，这就是语言的可扩展性。

XML 是从标准通用标记语言（SGML）中简化修改出来的，主要用到的有可扩展标记语言（XML）、可扩展样式语言（XSL）、XBRL 和 XPath 等。

XML 是从 1995 年开始有其雏形，并向 W3C（万维网联盟）提案，并在 1998 年二月发布为 W3C 的标准（XML1.0）。

XML 的前身 GML（The Standard Generalized Markup Language）是自 IBM 从 1960 年代就开始发展的 GML（Generalized Markup Language）标准化后的名称，SGML 的问题在于其本身是一种非常严谨的文件描述法，导致过于庞大复杂，难以理解和学习，进而影响其推广与应用。

GML 提出了两个重要概念，分别是：

- 文件中能够明确的将标示与内容分开
- 所有文件的标示使用方法均一致

1978 年，ANSI 将 GML 加以整理规范，发布成为 SGML，1986 年起为 ISO 所采用（ISO 8879），并且被广泛地运用在各种大型的文件计划中。

对于 SGML 的竞争对手 HTML，W3C 也提到了 HTML 的问题：

- 不能解决所有解释数据的问题（例如影音文件或化学公式、音乐符号等其他形态的内容）。
- 性能问题 - 需要下载整份文件，才能开始对文件做搜索。
- 扩充性、弹性、易读性均不佳。

为了解决以上问题，专家们使用 SGML 精简制作，并依照 HTML 的发展经验，产生出

一套使用上规则严谨但是简单的描述数据语言——XML。

XML 是在一个这样的背景下诞生的——为了有一个更中立的方式，让消费端自行决定要如何消化、呈现从服务端所提供的信息，后来 XML 被广泛用来作为跨平台之间交互数据的形式（主要针对数据的内容），通过不同的格式化描述手段（XSLT，CSS 等）可以完成最终的形式表达（生成对应的 HTML，PDF 或者其他的文件格式）。

和 HTML 相比，XML 被设计用来传送及携带数据信息，不用来表现或展示数据，HTML 语言则用来表现数据，所以 XML 用途的焦点是它说明数据是什么，以及携带数据信息。

下面是 RecipeBook（一种基于 XML 语法上的烹饪技术书刊）的例子，此标签可转换为 HTML、PDF 以及 Rich Text Format 并使用编程语言或 XSL。

```
<?xml ==?>version"1.0" encoding"UTF-8" "no"standalone=
<!DOCTYPE recipe PUBLIC "-//Happy-Monkey//DTD RecipeBook//EN
"http://www.happy-monkey.net/recipebook/recipebook.dtd">

<recipe>

<title>Peanutbutter On A Spoon</title>

<ingredientlist>
  <ingredient>Peanutbutter</ingredient>
</ingredientlist>

<preparation>Stick a spoon in a jar of peanutbutter, scoop
and pull out a big glob of peanutbutter.</preparation>

</recipe>
```

图 119.1: RecipeBook 示例

- 富文本（Rich Documents）- 自定文件描述并使其更丰富，例如属于文件为主的 XML 技术应用中，标记用来定义一份资料应该如何呈现。
- 元数据（Metadata）- 描述其它文件或网络信息，例如属于资料为主的 XML 技术应用中，标记是用来说明一份资料的意义。
- 配置文档（Configuration Files）- 描述软件设置的参数

下面的示例为小张发送给大元的便条（存储为 XML），可以看出 XML 定义结构、存储信息、传送信息，XML 文档仅是纯粹的信息标签，这些标签意义的展开依赖于应用它的程序。

```
<? xml version = “1.0” ? >
<小纸条>
  <收件人>大元</
  <发件人>小张</发件人>
  <主题>问候</主题>
  <具体内容>早啊，饭吃了没？ </具体内容>
</小纸条>
```

每个 XML 文档都由 XML 序言开始，在前面的代码中的第一行就是 XML 序言，`<?xml version="1.0"?>`，这一行代码会告诉解析器或浏览器这个文件应该按照 XML 规则进行解析。

不过，根元素到底叫 < 小纸条 > 还是 <Book> 则是由文档类型定义（DTD）或 XML 纲要定义的。如果 DTD 规定根元素必须叫 < 小便条 >，那么写作 < 小纸条 > 就不匹配要求。这种不匹配 DTD 或 XML 纲要的要求的 XML 文档，被称作不合法的 XML，反之则是合法的

XML。

XML 文件的第二行并不一定要包含文档元素；如果有注释或者其他内容，文档元素可以迟些出现。

这个 PI 一般会直接放在 XML 序言之后，通常由 Web 浏览器使用，来将 XML 数据以特殊的样式显示出来。

注意，XML 的结构有一个缺陷，那就是不支持分帧（framing）。例如，当多条 XML 消息在 TCP 上传输的时候，无法基于 XML 协议来确定一条 XML 消息是否已经结束。

119.1 DOM

119.2 libxml

119.3 SDO

119.4 SDO-DAS-Relational

119.5 SDO-DAS-XML

119.6 SimpleXML

119.7 WDDX

119.8 XMLDiff

119.9 XML Parser

119.10 XMLReader

119.11 XMLWriter

119.12 XSL

Chapter 120

DOM

120.1 Overview

DOM 扩展允许开发者通过 DOM API 来操作 XML 文档，PHP 默认启用 DOM 扩展（可以通过`--disable-dom` 禁用）。

DOM 默认使用 UTF-8 编码，开发者可以使用 utf8_encode() 和 utf8_decode() 来处理 ISO-8859-1 编码的文本或者使用 iconv 扩展来处理其他编码格式。

下面的 DOM 示例使用的 book.xml 示例文件如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
 "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" [
]>
<book id="listing">
 <title>My lists</title>
 <chapter id="books">
  <title>My books</title>
  <para>
   <informaltable>
    <tgroup cols="4">
     <thead>
      <row>
       <entry>Title</entry>
       <entry>Author</entry>
       <entry>Language</entry>
       <entry>ISBN</entry>
      </row>
     </thead>
```

```
    <tbody>
     <row>
      <entry>The Grapes of Wrath</entry>
      <entry>John Steinbeck</entry>
      <entry>en</entry>
      <entry>0140186409</entry>
     </row>
     <row>
      <entry>The Pearl</entry>
      <entry>John Steinbeck</entry>
      <entry>en</entry>
      <entry>014017737X</entry>
     </row>
     <row>
      <entry>Samarcande</entry>
      <entry>Amine Maalouf</entry>
      <entry>fr</entry>
      <entry>2253051209</entry>
     </row>
     <!-- TODO: I have a lot of remaining books to add.. -->
    </tbody>
   </tgroup>
  </informaltable>
 </para>
</chapter>
</book>
```

120.1.1 Requirement

DOM 和 SimpleXML 都需要 libxml 扩展的支持，因此在编译 PHP 时需要指定`--enable-libxml`，不过实际上 libxml 默认是开启的。

```
$ php --ri libxml
libxml

libXML support => active
libXML Compiled Version => 2.9.1
libXML Loaded Version => 20901
libXML streams => enabled
```

120.1.2 Resource Type

DOM 没有定义资源类型。

120.1.3 Build-in Module

DOM 默认为启用，可以在编译时通过如下选项禁用`--disable-dom`。

如果需要手动安装 XML 扩展，可以执行：

```
$ sudo apt-get install php7.0-xml
```

120.1.4 Runtime Configure

DOM 没有在 php.ini 中定义配置指令。

120.1.5 Predefined Constants

DOM 提供了自己的预定义常量，并且仅在 DOM 扩展编译入 PHP 或者在运行时动态载入 DOM 扩展时可用。

表 120.1: DOM 扩展的预定义常量

常量名	常量值	说明
XML_ELEMENT_NODE (integer)	1	Node is a DOMElement
XML_ATTRIBUTE_NODE (integer)	2	Node is a DOMAttr
XML_TEXT_NODE (integer)	3	Node is a DOMText
XML_CDATA_SECTION_NODE (integer)	4	Node is a DOMCharacterData
XML_ENTITY_REF_NODE (integer)	5	Node is a DOMEntityReference
XML_ENTITY_NODE (integer)	6	Node is a DOMEntity
XML_PI_NODE (integer)	7	Node is a DOMProcessingInstruction
XML_COMMENT_NODE (integer)	8	Node is a DOMComment
XML_DOCUMENT_NODE (integer)	9	Node is a DOMDocument
XML_DOCUMENT_TYPE_NODE (integer)	10	Node is a DOMDocumentType
XML_DOCUMENT_FRAG_NODE (integer)	11	Node is a DOMDocumentFragment
XML_NOTATION_NODE (integer)	12	Node is a DOMNotation
XML_HTML_DOCUMENT_NODE (integer)	13	
XML_DTD_NODE (integer)	14	
XML_ELEMENT_DECL_NODE (integer)	15	
XML_ATTRIBUTE_DECL_NODE (integer)	16	
XML_ENTITY_DECL_NODE (integer)	17	
XML_NAMESPACE_DECL_NODE (integer)	18	
XML_ATTRIBUTE_CDATA (integer)	1	
XML_ATTRIBUTE_ID (integer)	2	
XML_ATTRIBUTE_IDREF (integer)	3	

常量名	常量值	说明
XML_ATTRIBUTE_IDREFS (integer)	4	
XML_ATTRIBUTE_ENTITY (integer)	5	
XML_ATTRIBUTE_NMTOKEN (integer)	7	
XML_ATTRIBUTE_NMTOKENS (integer)	8	
XML_ATTRIBUTE_ENUMERATION (integer)	9	
XML_ATTRIBUTE_NOTATION (integer)	10	

表 120.2: DOMException 的预定义常量

常量名	常量值	说明
DOM_PHP_ERR (integer)	0	Error code not part of the DOM specification. Meant for PHP errors.
DOM_INDEX_SIZE_ERR (integer)	1	If index or size is negative, or greater than the allowed value.
DOMSTRING_SIZE_ERR (integer)	2	If the specified range of text does not fit into a DOMString.
DOM_HIERARCHY_REQUEST_ERR (integer)	3	If any node is inserted somewhere it doesn't belong
DOM_WRONG_DOCUMENT_ERR (integer)	4	If a node is used in a different document than the one that created it.
DOM_INVALID_CHARACTER_ERR (integer)	5	If an invalid or illegal character is specified, such as in a name.
DOM_NO_DATA_ALLOWED_ERR (integer)	6	If data is specified for a node which does not support data.
DOM_NO_MODIFICATION_ALLOWED_ERR (integer)	7	If an attempt is made to modify an object where modifications are not allowed.
DOM_NOT_FOUND_ERR (integer)	8	If an attempt is made to reference a node in a context where it does not exist.
DOM_NOT_SUPPORTED_ERR (integer)	9	If the implementation does not support the requested type of object or operation.
DOM_INUSE_ATTRIBUTE_ERR (integer)	10	If an attempt is made to add an attribute that is already in use elsewhere.
DOM_INVALID_STATE_ERR (integer)	11	If an attempt is made to use an object that is not, or is no longer, usable.
DOM_SYNTAX_ERR (integer)	12	If an invalid or illegal string is specified.
DOM_INVALID_MODIFICATION_ERR (integer)	13	If an attempt is made to modify the type of the underlying object.
DOM_NAMESPACE_ERR (integer)	14	If an attempt is made to create or change an object in a way which is incorrect with regard to namespaces.
DOM_INVALID_ACCESS_ERR (integer)	15	If a parameter or an operation is not supported by the underlying object.
DOM_VALIDATION_ERR (integer)	16	If a call to a method such as insertBefore or removeChild would make the Node invalid with respect to "partial validity", this exception would be raised and the operation would not be done.

120.2 DOMAttr

DOMAttr 表示 DOMElement 对象中的一个属性。

```
DOMAttr extends DOMNode {
    /* 属性 */
    public readonly string $name ;
    public readonly DOMElement $ownerElement ;
    public readonly bool $schemaTypeInfo ;
    public readonly bool $specified ;
    public string $value ;
    /* 方法 */
    public __construct ( string $name [, string $value ] )
    public bool isId ( void )
    /* 继承的方法 */
    public DOMNode DOMNode::appendChild ( DOMNode $newnode )
    public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
         [, array $ns_prefixes ]]]] )
    public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
        [, array $xpath [, array $ns_prefixes ]]]] )
    public DOMNode DOMNode::cloneNode ([ bool $deep ] )
    public int DOMNode::getLineNo ( void )
    public string DOMNode::getNodePath ( void )
    public bool DOMNode::hasAttributes ( void )
    public bool DOMNode::hasChildNodes ( void )
    public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
    public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
    public bool DOMNode::isSameNode ( DOMNode $node )
    public bool DOMNode::isSupported ( string $feature , string $version )
    public string DOMNode::lookupNamespaceURI ( string $prefix )
    public string DOMNode::lookupPrefix ( string $namespaceURI )
    public void DOMNode::normalize ( void )
    public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.2.1 $name

$name 是 XML 属性的名称

120.2.2 $ownerElement

$ownerElement 是包含 XML 属性的 XML 元素

120.2.3 $schemaTypeInfo

$schemaTypeInfo 未实现，总是返回 NULL。

120.2.4 $specified

$specified 未实现，总是返回 NULL。

120.2.5 $value

$value 是 XML 属性的值。

120.2.6 DOMAttr::__construct()

创建一个新的只读的 DOMAttr 对象。

```
public DOMAttr::__construct ( string $name [, string $value ] )
```

创建新的 DOMAttr 对象，而且这个 DOMAttr 对象是只读的，可以被附加到 XML 文档，但是额外的节点直到该节点与 XML 文档相关联时才可以不附加到该节点。

要创建可写节点，可以使用 DOMDocument::createAttribute()。

- $name - 属性的 tag 名字
- $value - 属性的值

```
<?php
$dom = new DOMDocument('1.0','iso-8859-1');
$element = $dom->appendChild(new DOMElement('root'));
$attr = $element->setAttributeNode(new DOMAttr('attr','attrvalue'));
echo $dom->saveXML();

// 以上例程会输出:
<?xml version="1.0" encoding="iso-8859-1"?>
<root attr="attrvalue" />
```

120.2.7 DOMAttr::isId()

检查一个 XML 属性是否是一个已定义的 ID。

```
public bool DOMAttr::isId ( void )
```

isId() 函数检查属性是否是定义的 ID，成功时返回 TRUE，或者在失败时返回 FALSE。

根据 DOM 标准，这需要将属性 ID 定义为 ID 类型的 DTD。在使用此函数之前，您需要使用 DOMDocument::validate 或 DOMDocument::validateOnParse 来验证 XML 文档。

```php
<?php
$doc = new DomDocument;
// validate xml document before refering to the id
$doc->validateOnParse = true;
$doc->Load('book.xml');

// retrieve the attribute named id of the chapter element
$attr = $doc->getElementsByTagName('chapter')->item(0)->getAttributeNode('id');
var_dump($attr->isId()); // bool(true)
```

120.3 DOMCdataSection

DOMCdataSection 继承自 DOMText，用于 CData 结构的文本化表示。

```
DOMCdataSection extends DOMText {
    /* Methods */
    public __construct ( string $value )
    /* Inherited methods */
    public bool DOMText::isWhitespaceInElementContent ( void )
    public DOMText DOMText::splitText ( int $offset )
}
```

120.3.1 DOMCdataSection::__construct()

构造新的 DOMCdataSection 对象

```
public DOMCdataSection::__construct ( string $value )
```

构造一个新的 CDATA 节点，其过程类似于 DOMText 类。

- $value - CDATA 节点的值。如果没有提供，那么会创建一个新的空的 CDATA 节点。

```php
<?php
$dom = new DOMDocument('1.0', 'utf-8');
$element = $dom->appendChild(new DOMElement('root'));
$text = $element->appendChild(new DOMCdataSection('root value'));
echo $dom->saveXML();
// 以上例程会输出：
// <?xml version="1.0" encoding="utf-8"?>
// <root><![CDATA[root value]]></root>
```

120.4 DOMCharacterData

表示具有字符数据的节点。没有节点直接对应这个类，但其他节点从它继承。

```
DOMCharacterData extends DOMNode {
    /* 属性 */
    public string $data ;
    readonly public int $length ;
    /* 方法 */
    void appendData ( string $data )
    void deleteData ( int $offset , int $count )
    void insertData ( int $offset , string $data )
    void replaceData ( int $offset , int $count , string $data )
    string substringData ( int $offset , int $count )
    /* 继承的方法 */
    public DOMNode DOMNode::appendChild ( DOMNode $newnode )
    public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
         [, array $ns_prefixes ]]]] )
    public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
        [, array $xpath [, array $ns_prefixes ]]]] )
    public DOMNode DOMNode::cloneNode ([ bool $deep ] )
    public int DOMNode::getLineNo ( void )
    public string DOMNode::getNodePath ( void )
    public bool DOMNode::hasAttributes ( void )
    public bool DOMNode::hasChildNodes ( void )
    public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
    public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
    public bool DOMNode::isSameNode ( DOMNode $node )
    public bool DOMNode::isSupported ( string $feature , string $version )
    public string DOMNode::lookupNamespaceURI ( string $prefix )
    public string DOMNode::lookupPrefix ( string $namespaceURI )
    public void DOMNode::normalize ( void )
    public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.4.1 $data

XML 节点的内容

120.4.2 $length

XML 节点内容的长度

120.4.3 DOMCharacterData::appendData()

将字符串附加到节点的字符数据的结尾

```
void DOMCharacterData::appendData ( string $data )
```

在 XML 节点的字符末尾追加 $data 变量中的字符串值。

- $data - 要追加的字符串

120.4.4 DOMCharacterData::deleteData()

从节点中删除一定范围的字符

```
void DOMCharacterData::deleteData ( int $offset , int $count )
```

从 XML 节点的 $offset 指定的偏移开始删除 $count 个字符。

- $offset - 开始删除字符的偏移。
- $count - 要删除的字符个数。

如果 $offset 和 $count 超过长度，那么节点中到末尾的数据都会删除。

如果 $offset 或 $count 是负数，或者大于数据中的 16 位单元的总长度，deleteData() 都会抛出 DOM_INDEX_SIZE_ERR 错误。

120.4.5 DOMCharacterData::insertData()

以指定的 16 位单位偏移插入一个字符串

```
void DOMCharacterData::insertData ( int $offset , string $data )
```

从 $offset 指定的位置开始插入字符串数据 $data。

- $offset - 开始插入字符串的偏移位置
- $data - 要插入的字符串

如果 $offset 或者大于数据中的 16 位单元的总长度，insertData() 都会抛出 DOM_INDEX_SIZE_ERR 错误。

120.4.6 DOMCharacterData::replaceData()

替换 DOMCharacterData 节点中的子字符串

```
void DOMCharacterData::replaceData ( int $offset , int $count , string $data )
```

从 $offset 指定的位置开始，使用 $data 的数据替换 $count 个字符。

- $offset - 开始替换字符的偏移位置
- $count - 要替换字符的个数
- $data - 要替换已有的的字符串数据

如果 $offset 和 $count 超过长度，那么节点中到末尾的数据都会被替换。

如果 $offset 或 $count 是负数或者大于数据中的 16 位单元的总长度，replaceData() 都会抛出 DOM_INDEX_SIZE_ERR 错误。

120.4.7 DOMCharacterData::substringData()

从节点提取一系列数据

```
string DOMCharacterData::substringData ( int $offset , int $count )
```

返回指定的子字符串。

- $offset - 需要提取子字符串的开始偏移位置
- $count - 需要提取子字符串的字符个数

如果 $offset 和 $count 超过长度，那么将会返回直到末尾的所有数据的 16 位单元。

如果 $offset 或 $count 是负数或者大于数据中的 16 位单元的总长度，substringData() 都会抛出 DOM_INDEX_SIZE_ERR 错误。

120.5 DOMComment

输出注释节点，由 <!-- 和--> 分隔的字符。

```
DOMComment extends DOMCharacterData {
   /* 方法 */
   public __construct ([ string $value ] )
   /* 继承的方法 */
   void DOMCharacterData::appendData ( string $data )
   void DOMCharacterData::deleteData ( int $offset , int $count )
   void DOMCharacterData::insertData ( int $offset , string $data )
   void DOMCharacterData::replaceData ( int $offset , int $count , string $data )
   string DOMCharacterData::substringData ( int $offset , int $count )
   public DOMNode DOMNode::appendChild ( DOMNode $newnode )
   public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
        [, array $ns_prefixes ]]]] )
   public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
       [, array $xpath [, array $ns_prefixes ]]]] )
   public DOMNode DOMNode::cloneNode ([ bool $deep ] )
   public int DOMNode::getLineNo ( void )
   public string DOMNode::getNodePath ( void )
```

```
    public bool DOMNode::hasAttributes ( void )
    public bool DOMNode::hasChildNodes ( void )
    public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
    public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
    public bool DOMNode::isSameNode ( DOMNode $node )
    public bool DOMNode::isSupported ( string $feature , string $version )
    public string DOMNode::lookupNamespaceURI ( string $prefix )
    public string DOMNode::lookupPrefix ( string $namespaceURI )
    public void DOMNode::normalize ( void )
    public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.5.1 DOMComment::__construct()

创建一个新的 DOMComment 对象。

```
public DOMComment::__construct ([ string $value ] )
```

创建新的 DOMComment 对象，而且对象是只读的，可以被附加到 XML 文档，但是附加节点可以不附加到该节点，直到该节点与文档相关联。

要创建可写节点，需要使用 DOMDocument::createComment()。

- $value - XML Comment 的值

```
<?php
$dom = new DOMDocument('1.0','iso-8859-1');
$element = $dom->appendChild(new DOMElement('root'));
$comment = $element->appendChild(new DOMComment('root comment'));
echo $dom->saveXML();
// 返回结果如下:
// <?xml version="1.0" encoding="iso-8859-1"?>
//  <root><!--root comment--></root>
```

120.6 DOMDocument

表示整个 HTML 或 XML 文档; 用作文档树的根。

```
DOMDocument extends DOMNode {
    /* 属性 */
    readonly public string $actualEncoding ;
    readonly public DOMConfiguration $config ;
    readonly public DOMDocumentType $doctype ;
```

```
readonly public DOMElement $documentElement ;
public string $documentURI ;
public string $encoding ;
public bool $formatOutput ;
readonly public DOMImplementation $implementation ;
public bool $preserveWhiteSpace = true ;
public bool $recover ;
public bool $resolveExternals ;
public bool $standalone ;
public bool $strictErrorChecking = true ;
public bool $substituteEntities ;
public bool $validateOnParse = false ;
public string $version ;
readonly public string $xmlEncoding ;
public bool $xmlStandalone ;
public string $xmlVersion ;
/* 方法 */
public __construct ([ string $version [, string $encoding ]] )
public DOMAttr createAttribute ( string $name )
public DOMAttr createAttributeNS ( string $namespaceURI , string $qualifiedName )
public DOMCDATASection createCDATASection ( string $data )
public DOMComment createComment ( string $data )
public DOMDocumentFragment createDocumentFragment ( void )
public DOMElement createElement ( string $name [, string $value ] )
public DOMElement createElementNS ( string $namespaceURI , string $qualifiedName [,
    string $value ] )
public DOMEntityReference createEntityReference ( string $name )
public DOMProcessingInstruction createProcessingInstruction ( string $target [,
    string $data ] )
public DOMText createTextNode ( string $content )
public DOMElement getElementById ( string $elementId )
public DOMNodeList getElementsByTagName ( string $name )
public DOMNodeList getElementsByTagNameNS ( string $namespaceURI , string $localName
     )
public DOMNode importNode ( DOMNode $importedNode [, bool $deep ] )
public mixed load ( string $filename [, int $options = 0 ] )
public bool loadHTML ( string $source [, int $options = 0 ] )
public bool loadHTMLFile ( string $filename [, int $options = 0 ] )
public mixed loadXML ( string $source [, int $options = 0 ] )
public void normalizeDocument ( void )
public bool registerNodeClass ( string $baseclass , string $extendedclass )
```

```
    public bool relaxNGValidate ( string $filename )
    public bool relaxNGValidateSource ( string $source )
    public int save ( string $filename [, int $options ] )
    public string saveHTML ([ DOMNode $node = NULL ] )
    public int saveHTMLFile ( string $filename )
    public string saveXML ([ DOMNode $node [, int $options ]] )
    public bool schemaValidate ( string $filename [, int $flags ] )
    public bool schemaValidateSource ( string $source [, int $flags ] )
    public bool validate ( void )
    public int xinclude ([ int $options ] )
    /* 继承的方法 */
    public DOMNode DOMNode::appendChild ( DOMNode $newnode )
    public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array
        $xpath [, array $ns_prefixes ]]]] )
    public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
         [, array $xpath [, array $ns_prefixes ]]]] )
    public DOMNode DOMNode::cloneNode ([ bool $deep ] )
    public int DOMNode::getLineNo ( void )
    public string DOMNode::getNodePath ( void )
    public bool DOMNode::hasAttributes ( void )
    public bool DOMNode::hasChildNodes ( void )
    public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
    public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
    public bool DOMNode::isSameNode ( DOMNode $node )
    public bool DOMNode::isSupported ( string $feature , string $version )
    public string DOMNode::lookupNamespaceURI ( string $prefix )
    public string DOMNode::lookupPrefix ( string $namespaceURI )
    public void DOMNode::normalize ( void )
    public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.6.1 $actualEncoding

Deprecated. Actual encoding of the document, is a readonly equivalent to encoding.

120.6.2 $config

Deprecated. Configuration used when DOMDocument::normalizeDocument() is invoked.

120.6.3 $doctype

The Document Type Declaration associated with this document.

120.6.4 $documentElement

This is a convenience attribute that allows direct access to the child node that is the document element of the document.

120.6.5 $documentURI

The location of the document or NULL if undefined.

120.6.6 $encoding

Encoding of the document, as specified by the XML declaration. This attribute is not present in the final DOM Level 3 specification, but is the only way of manipulating XML document encoding in this implementation.

120.6.7 $formatOutput

Nicely formats output with indentation and extra space.

120.6.8 $implementation

The DOMImplementation object that handles this document.

120.6.9 $preserveWhiteSpace

Do not remove redundant white space. Default to TRUE.

120.6.10 $recover

Proprietary. Enables recovery mode, i.e. trying to parse non-well formed documents. This attribute is not part of the DOM specification and is specific to libxml.

120.6.11 $resolveExternals

Set it to TRUE to load external entities from a doctype declaration. This is useful for including character entities in your XML document.

120.6.12 $standalone

Deprecated. Whether or not the document is standalone, as specified by the XML declaration, corresponds to xmlStandalone.

120.6.13 $strictErrorChecking

Throws DOMException on errors. Default to TRUE.

120.6.14 $substituteEntities

Proprietary. Whether or not to substitute entities. This attribute is not part of the DOM specification and is specific to libxml.

120.6.15 $validateOnParse

Loads and validates against the DTD. Default to FALSE.

120.6.16 $version

Deprecated. Version of XML, corresponds to xmlVersion.

120.6.17 $xmlEncoding

An attribute specifying, as part of the XML declaration, the encoding of this document. This is NULL when unspecified or when it is not known, such as when the Document was created in memory.

120.6.18 $xmlStandalone

An attribute specifying, as part of the XML declaration, whether this document is standalone. This is FALSE when unspecified.

120.6.19 $xmlVersion

An attribute specifying, as part of the XML declaration, the version number of this document. If there is no declaration and if this document supports the ”XML” feature, the value is ”1.0”.

120.6.20 DOMDocument::__construct()

120.6.21 DOMDocument::createAttribute()

120.6.22 DOMDocument::createAttributeNS()

120.6.23 DOMDocument::createCDATASection()

120.6.24 DOMDocument::createComment()

120.6.25 DOMDocument::createDocumentFragment()

120.6.26 DOMDocument::createElement()

120.6.27 DOMDocument::createElementNS()

120.6.28 DOMDocument::createEntityReference()

120.6.29 DOMDocument::createProcessingInstruction()

120.6.30 DOMDocument::createTextNode()

120.6.31 DOMDocument::getElementById()

120.6.32 DOMDocument::getElementsByTagName()

120.6.33 DOMDocument::getElementsByTagNameNS()

120.6.34 DOMDocument::importNode()

120.6.35 DOMDocument::load()

120.6.36 DOMDocument::loadHTML()

120.6.37 DOMDocument::loadHTMLFile()

120.6.38 DOMDocument::normalizeDocument()

120.6.39 DOMDocument::registerNodeClass()

120.6.40 DOMDocument::relaxNGValidate()

120.6.41 DOMDocument::relaxNGValidateSource()

120.6.42 DOMDocument::save()

120.6.43 DOMDocument::saveHTML()

120.6.44 DOMDocument::saveHTMLFile()

120.6.45 DOMDocument::saveXML()

120.6.46 DOMDocument::schemaValidate()

120.6.47 DOMDocument::schemaValidateSource()

120.6.48 DOMDocument::validate()

120.6.49 DOMDocument::xinclude()

120.7 DOMDocumentFragment

```
DOMDocumentFragment extends DOMNode {
   /* 属性 */
   /* 方法 */
   public bool appendXML ( string $data )
   /* 继承的方法 */
   public DOMNode DOMNode::appendChild ( DOMNode $newnode )
   public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
        [, array $ns_prefixes ]]]] )
   public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
       [, array $xpath [, array $ns_prefixes ]]]] )
   public DOMNode DOMNode::cloneNode ([ bool $deep ] )
   public int DOMNode::getLineNo ( void )
   public string DOMNode::getNodePath ( void )
   public bool DOMNode::hasAttributes ( void )
   public bool DOMNode::hasChildNodes ( void )
   public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
   public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
   public bool DOMNode::isSameNode ( DOMNode $node )
   public bool DOMNode::isSupported ( string $feature , string $version )
   public string DOMNode::lookupNamespaceURI ( string $prefix )
   public string DOMNode::lookupPrefix ( string $namespaceURI )
   public void DOMNode::normalize ( void )
   public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
```

```
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.7.1 DOMDocumentFragment::appendXML()

追加原始 XML 数据。

```
public bool DOMDocumentFragment::appendXML ( string $data )
```

将原始 XML 数据附加到 DOMDocumentFragment，成功时返回 TRUE，或者在失败时返回 FALSE。

此方法不是 DOM 标准的一部分。它被创建为一个更简单的方法在 DOMDocument 中附加一个 XML DocumentFragment。

如果需要遵循标准，必须创建一个临时 DOMDocument 与一个虚拟根，然后循环通过 XML 数据根的子节点来附加它们。

- $data - 要追加的 XML 数据

```
<?php
$doc = new DOMDocument();
$doc->loadXML('<root/>');
$f = $doc->createDocmentFragment();
$f->appendXML('<foo>text1</foo><bar>text2</bar>');
$doc->documentElement->appendChild($f);
echo $doc->saveXML();
// 以上例程会输出:
// <?xml version="1.0"?>
// <root><foo>text1</foo><bar>text2</bar></root>
```

120.8 DOMDocumentType

每个 DOMDocument 都有一个 doctype 属性，其值为 NULL 或 DOMDocumentType 对象。

```
DOMDocumentType extends DOMNode {
    /* 属性 */
    readonly public string $publicId ;
    readonly public string $systemId ;
    readonly public string $name ;
    readonly public DOMNamedNodeMap $entities ;
    readonly public DOMNamedNodeMap $notations ;
    readonly public string $internalSubset ;
    /* 继承的方法 */
```

```
    public DOMNode DOMNode::appendChild ( DOMNode $newnode )
    public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
         [, array $ns_prefixes ]]]] )
    public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
        [, array $xpath [, array $ns_prefixes ]]]] )
    public DOMNode DOMNode::cloneNode ([ bool $deep ] )
    public int DOMNode::getLineNo ( void )
    public string DOMNode::getNodePath ( void )
    public bool DOMNode::hasAttributes ( void )
    public bool DOMNode::hasChildNodes ( void )
    public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
    public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
    public bool DOMNode::isSameNode ( DOMNode $node )
    public bool DOMNode::isSupported ( string $feature , string $version )
    public string DOMNode::lookupNamespaceURI ( string $prefix )
    public string DOMNode::lookupPrefix ( string $namespaceURI )
    public void DOMNode::normalize ( void )
    public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
    public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.8.1 $publicId

120.8.2 $systemId

120.8.3 $name

120.8.4 $entities

120.8.5 $notations

120.8.6 $internalSubset

120.9 DOMElement

```
DOMElement extends DOMNode {
  /* 属性 */
  readonly public bool $schemaTypeInfo ;
  readonly public string $tagName ;
  /* 方法 */
  public __construct ( string $name [, string $value [, string $namespaceURI ]] )
  public string getAttribute ( string $name )
  public DOMAttr getAttributeNode ( string $name )
  public DOMAttr getAttributeNodeNS ( string $namespaceURI , string $localName )
  public string getAttributeNS ( string $namespaceURI , string $localName )
  public DOMNodeList getElementsByTagName ( string $name )
  public DOMNodeList getElementsByTagNameNS ( string $namespaceURI , string $localName
      )
  public bool hasAttribute ( string $name )
  public bool hasAttributeNS ( string $namespaceURI , string $localName )
  public bool removeAttribute ( string $name )
  public bool removeAttributeNode ( DOMAttr $oldnode )
  public bool removeAttributeNS ( string $namespaceURI , string $localName )
  public DOMAttr setAttribute ( string $name , string $value )
  public DOMAttr setAttributeNode ( DOMAttr $attr )
  public DOMAttr setAttributeNodeNS ( DOMAttr $attr )
  public void setAttributeNS ( string $namespaceURI , string $qualifiedName , string
      $value )
  public void setIdAttribute ( string $name , bool $isId )
  public void setIdAttributeNode ( DOMAttr $attr , bool $isId )
  public void setIdAttributeNS ( string $namespaceURI , string $localName , bool $isId
      )
  /* 继承的方法 */
  public DOMNode DOMNode::appendChild ( DOMNode $newnode )
  public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath
       [, array $ns_prefixes ]]]] )
  public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments
      [, array $xpath [, array $ns_prefixes ]]]] )
  public DOMNode DOMNode::cloneNode ([ bool $deep ] )
  public int DOMNode::getLineNo ( void )
  public string DOMNode::getNodePath ( void )
  public bool DOMNode::hasAttributes ( void )
```

```
  public bool DOMNode::hasChildNodes ( void )
  public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
  public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
  public bool DOMNode::isSameNode ( DOMNode $node )
  public bool DOMNode::isSupported ( string $feature , string $version )
  public string DOMNode::lookupNamespaceURI ( string $prefix )
  public string DOMNode::lookupPrefix ( string $namespaceURI )
  public void DOMNode::normalize ( void )
  public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
  public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.9.1 $schemaTypeInfo

120.9.2 $tagName

120.9.3 DOMElement::__construct()

120.9.4 DOMElement::getAttribute()

120.9.5 DOMElement::getAttributeNode()

120.9.6 DOMElement::getAttributeNodeNS()

120.9.7 DOMElement::getElementsByTagName()

120.9.8 DOMElement::getElementsByTagNameNS()

120.9.9 DOMElement::hasAttribute()

120.9.10 DOMElement::hasAttributeNS()

120.9.11 DOMElement::removeAttribute()

120.9.12 DOMElement::removeAttributeNode()

120.9.13 DOMElement::removeAttributeNS()

120.9.14 DOMElement::setAttribute()

120.9.15 DOMElement::setAttributeNode()

120.9.16 DOMDocument::setAttributeNodeNS()

120.9.17 DOMDocument::setAttributeNS()

120.9.18 DOMDocument::setIdAttribute()

120.9.19 DOMDocument::setIdAttributeNode()

120.9.20 DOMDocument::setIdAttributeNS()

120.10 DOMEntity

DOMEntity 接口表示 XML 文档中已解析或未解析的已知实体。

```
DOMEntity extends DOMNode {
/* 属性 */
readonly public string $publicId ;
readonly public string $systemId ;
readonly public string $notationName ;
public string $actualEncoding ;
readonly public string $encoding ;
readonly public string $version ;
/* 继承的方法 */
public DOMNode DOMNode::appendChild ( DOMNode $newnode )
public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [,
     array $ns_prefixes ]]]] )
public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [,
    array $xpath [, array $ns_prefixes ]]]] )
public DOMNode DOMNode::cloneNode ([ bool $deep ] )
public int DOMNode::getLineNo ( void )
public string DOMNode::getNodePath ( void )
public bool DOMNode::hasAttributes ( void )
public bool DOMNode::hasChildNodes ( void )
public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
public bool DOMNode::isSameNode ( DOMNode $node )
public bool DOMNode::isSupported ( string $feature , string $version )
public string DOMNode::lookupNamespaceURI ( string $prefix )
public string DOMNode::lookupPrefix ( string $namespaceURI )
public void DOMNode::normalize ( void )
public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.10.1 $pupblidId

120.10.2 $systemId

120.10.3 $notationName

120.10.4 $actualEncoding

120.10.5 $encoding

120.10.6 $version

120.11 DOMEntityReference

```
DOMEntityReference extends DOMNode {
/* 属性 */
/* 方法 */
public __construct ( string $name )
/* 继承的方法 */
public DOMNode DOMNode::appendChild ( DOMNode $newnode )
public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [,
     array $ns_prefixes ]]]] )
public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [,
    array $xpath [, array $ns_prefixes ]]]] )
public DOMNode DOMNode::cloneNode ([ bool $deep ] )
public int DOMNode::getLineNo ( void )
public string DOMNode::getNodePath ( void )
public bool DOMNode::hasAttributes ( void )
public bool DOMNode::hasChildNodes ( void )
public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
public bool DOMNode::isSameNode ( DOMNode $node )
public bool DOMNode::isSupported ( string $feature , string $version )
public string DOMNode::lookupNamespaceURI ( string $prefix )
public string DOMNode::lookupPrefix ( string $namespaceURI )
public void DOMNode::normalize ( void )
public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.11.1 DOMEntityReference::__construct()

120.12 DOMException

DOM 操作在特定情况下（即当由于逻辑原因而不可能执行操作时）引发 DOMException 异常。

```
DOMException extends Exception {
/* 属性 */
readonly public int $code ;
/* 继承的方法 */
final public string Exception::getMessage ( void )
final public Exception Exception::getPrevious ( void )
final public int Exception::getCode ( void )
final public string Exception::getFile ( void )
final public int Exception::getLine ( void )
final public array Exception::getTrace ( void )
final public string Exception::getTraceAsString ( void )
public string Exception::__toString ( void )
final private void Exception::__clone ( void )
}
```

120.12.1 $code

120.13 DOMImplementation

DOMImplementation 接口提供了许多用于执行独立于文档对象模型的任何特定实例的操作的方法。

```
DOMImplementation {
/* 属性 */
/* 方法 */
__construct ( void )
public DOMDocument createDocument ([ string $namespaceURI = NULL [, string
    $qualifiedName = NULL [, DOMDocumentType $doctype = NULL ]]] )
public DOMDocumentType createDocumentType ([ string $qualifiedName = NULL [, string
    $publicId = NULL [, string $systemId = NULL ]]] )
public bool hasFeature ( string $feature , string $version )
}
```

120.13.1 DOMImplementation::__construct()

120.13.2 DOMImplementation::createDocument()

120.13.3 DOMImplementation::createDocument()

120.13.4 DOMImplementation::createDocumentType()

120.13.5 DOMImplementation::hasFeature()

120.14 DOMNamedNodeMap

```
DOMNamedNodeMap implements Traversable {
/* 属性 */
readonly public int $length ;
/* 方法 */
DOMNode getNamedItem ( string $name )
DOMNode getNamedItemNS ( string $namespaceURI , string $localName )
DOMNode item ( int $index )
}
```

120.14.1 $length

120.14.2 DOMNamedNodeMap::getNamedItem()

120.14.3 DOMNamedNodeMap::getNamedItemNS()

120.14.4 DOMNamedNodeMap::item()

120.15 DOMNode

```
DOMNode {
/* 属性 */
public readonly string $nodeName ;
public string $nodeValue ;
public readonly int $nodeType ;
public readonly DOMNode $parentNode ;
public readonly DOMNodeList $childNodes ;
public readonly DOMNode $firstChild ;
public readonly DOMNode $lastChild ;
public readonly DOMNode $previousSibling ;
public readonly DOMNode $nextSibling ;
public readonly DOMNamedNodeMap $attributes ;
public readonly DOMDocument $ownerDocument ;
public readonly string $namespaceURI ;
public string $prefix ;
public readonly string $localName ;
public readonly string $baseURI ;
public string $textContent ;
/* 方法 */
public DOMNode appendChild ( DOMNode $newnode )
public string C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [, array
    $ns_prefixes ]]]] )
public int C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [, array
    $xpath [, array $ns_prefixes ]]]] )
```

```
public DOMNode cloneNode ([ bool $deep ] )
public int getLineNo ( void )
public string getNodePath ( void )
public bool hasAttributes ( void )
public bool hasChildNodes ( void )
public DOMNode insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool isDefaultNamespace ( string $namespaceURI )
public bool isSameNode ( DOMNode $node )
public bool isSupported ( string $feature , string $version )
public string lookupNamespaceURI ( string $prefix )
public string lookupPrefix ( string $namespaceURI )
public void normalize ( void )
public DOMNode removeChild ( DOMNode $oldnode )
public DOMNode replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.15.1 $nodeName

Returns the most accurate name for the current node type

120.15.2 $nodeValue

The value of this node, depending on its type. Contrary to the W3C specification, the node value of DOMElement nodes is equal to DOMNode::textContent instead of NULL.

120.15.3 $nodeType

Gets the type of the node. One of the predefined XML_xxx_NODE constants

120.15.4 $parentNode

The parent of this node. If there is no such node, this returns NULL.

120.15.5 $childNodes

A DOMNodeList that contains all children of this node. If there are no children, this is an empty DOMNodeList.

120.15.6 $firstChild

The first child of this node. If there is no such node, this returns NULL.

120.15.7 $lastChild

The last child of this node. If there is no such node, this returns NULL.

120.15.8 $previousSibling

The node immediately preceding this node. If there is no such node, this returns NULL.

120.15.9 $nextSibling

The node immediately following this node. If there is no such node, this returns NULL.

120.15.10 $attributes

A DOMNamedNodeMap containing the attributes of this node (if it is a DOMElement) or NULL otherwise.

120.15.11 $ownerDocument

The DOMDocument object associated with this node.

120.15.12 $namespaceURI

The namespace URI of this node, or NULL if it is unspecified.

120.15.13 $prefix

The namespace prefix of this node, or NULL if it is unspecified.

120.15.14 $localName

Returns the local part of the qualified name of this node.

120.15.15 $baseURI

The absolute base URI of this node or NULL if the implementation wasn't able to obtain an absolute URI.

120.15.16 $textContent

The text content of this node and its descendants.

120.15.17 DOMNode::appendChild()

120.15.18 DOMNode::C14N()

120.15.19 DOMNode::C14NFile()

120.15.20 DOMNode::cloneNode()

120.15.21 DOMNode::getLineNo()

120.15.22 DOMNode::getNodePath()

120.15.23 DOMNode::hasAttributes()

120.15.24 DOMNode::hasChildNodes()

120.15.25 DOMNode::insertBefore()

120.15.26 DOMNode::isDefaultNamespace()

120.15.27 DOMNode::isSameNode()

120.15.28 DOMNode::isSupported()

120.15.29 DOMNode::lookupNamespaceURI()

120.15.30 DOMNode::lookupPrefix()

120.15.31 DOMNode::normalize()

120.15.32 DOMNode::removeChild()

120.15.33 DOMNode::replaceChild()

120.16 DOMNodeList

```
DOMNodeList implements Traversable {
/* 属性 */
readonly public int $length ;
/* 方法 */
DOMElement DOMNodelist::item ( int $index )
}
```

120.16.1 $length

120.16.2 DOMNodelist::item()

120.17 DOMNotation

```
DOMNotation extends DOMNode {
/* 属性 */
readonly public string $publicId ;
readonly public string $systemId ;
/* 继承的方法 */
public DOMNode DOMNode::appendChild ( DOMNode $newnode )
public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [,
     array $ns_prefixes ]]]] )
public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [,
    array $xpath [, array $ns_prefixes ]]]] )
public DOMNode DOMNode::cloneNode ([ bool $deep ] )
public int DOMNode::getLineNo ( void )
public string DOMNode::getNodePath ( void )
public bool DOMNode::hasAttributes ( void )
public bool DOMNode::hasChildNodes ( void )
public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
public bool DOMNode::isSameNode ( DOMNode $node )
public bool DOMNode::isSupported ( string $feature , string $version )
public string DOMNode::lookupNamespaceURI ( string $prefix )
public string DOMNode::lookupPrefix ( string $namespaceURI )
public void DOMNode::normalize ( void )
public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.17.1 $publicId

120.17.2 $systemId

120.18 DOMProcessingInstruction

```
DOMProcessingInstruction extends DOMNode {
/* 属性 */
readonly public string $target ;
public string $data ;
/* 方法 */
public __construct ( string $name [, string $value ] )
/* 继承的方法 */
public DOMNode DOMNode::appendChild ( DOMNode $newnode )
public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [,
     array $ns_prefixes ]]]] )
public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [,
    array $xpath [, array $ns_prefixes ]]]] )
public DOMNode DOMNode::cloneNode ([ bool $deep ] )
public int DOMNode::getLineNo ( void )
public string DOMNode::getNodePath ( void )
public bool DOMNode::hasAttributes ( void )
public bool DOMNode::hasChildNodes ( void )
public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
public bool DOMNode::isSameNode ( DOMNode $node )
public bool DOMNode::isSupported ( string $feature , string $version )
public string DOMNode::lookupNamespaceURI ( string $prefix )
public string DOMNode::lookupPrefix ( string $namespaceURI )
public void DOMNode::normalize ( void )
public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.18.1 $target

120.18.2 $data

120.18.3 DOMProcessingInstruction::__construct()

120.19 DOMText

DOMText 类继承自 DOMCharacterData，可以用来表示 DOMElement 或 DOMAttr 的文本内容。

```
DOMText extends DOMCharacterData {
/* 属性 */
readonly public string $wholeText ;
/* 方法 */
public __construct ([ string $value ] )
public bool isWhitespaceInElementContent ( void )
public DOMText splitText ( int $offset )
/* 继承的方法 */
public DOMNode DOMNode::appendChild ( DOMNode $newnode )
public string DOMNode::C14N ([ bool $exclusive [, bool $with_comments [, array $xpath [,
     array $ns_prefixes ]]]] )
public int DOMNode::C14NFile ( string $uri [, bool $exclusive [, bool $with_comments [,
    array $xpath [, array $ns_prefixes ]]]] )
public DOMNode DOMNode::cloneNode ([ bool $deep ] )
public int DOMNode::getLineNo ( void )
public string DOMNode::getNodePath ( void )
public bool DOMNode::hasAttributes ( void )
public bool DOMNode::hasChildNodes ( void )
public DOMNode DOMNode::insertBefore ( DOMNode $newnode [, DOMNode $refnode ] )
public bool DOMNode::isDefaultNamespace ( string $namespaceURI )
public bool DOMNode::isSameNode ( DOMNode $node )
public bool DOMNode::isSupported ( string $feature , string $version )
public string DOMNode::lookupNamespaceURI ( string $prefix )
public string DOMNode::lookupPrefix ( string $namespaceURI )
public void DOMNode::normalize ( void )
public DOMNode DOMNode::removeChild ( DOMNode $oldnode )
public DOMNode DOMNode::replaceChild ( DOMNode $newnode , DOMNode $oldnode )
}
```

120.19.1 $wholeText

120.19.2 DOMText::__construct()

120.19.3 DOMText::isWhitespaceInElementContent()

120.19.4 DOMText::splitText()

120.20 DOMXPath

DOMXPath 支持 XPath 1.0。

```
DOMXPath {
/* 属性 */
public DOMDocument $document ;
/* 方法 */
public __construct ( DOMDocument $doc )
public mixed evaluate ( string $expression [, DOMNode $contextnode [, bool
    $registerNodeNS = true ]] )
public DOMNodeList query ( string $expression [, DOMNode $contextnode [, bool
    $registerNodeNS = true ]] )
public bool registerNamespace ( string $prefix , string $namespaceURI )
public void registerPhpFunctions ([ mixed $restrict ] )
}
```

120.20.1 $document

120.20.2 DOMXPath::__construct()

120.20.3 DOMXPath::evaluate()

120.20.4 DOMXPath::query()

120.20.5 DOMXPath::registerNamespace()

120.20.6 DOMXPath::registerPhpFunctions()

120.21 DOM Functions

120.21.1 dom_import_simplexml()

dom_import_simplexml() 可以从一个 SimpleXML 对象获取一个 DOMElement 对象。

```
DOMElement dom_import_simplexml ( SimpleXMLElement $node )
```

这里的 $node 是 SimpleXMLElement 对象的节点，返回添加的 DOMElement 节点，发生错误则返回 FALSE。

dom_import_simplexml() 函数接受 SimpleXML 类的节点节点，并将其转换为 DOMElement 节点，然后这个新对象可以用作原生 DOMElement 节点。

```
<?php
$sxe = simplexml_load_string('<books><book><title>blah</title></book></books>');

if ($sxe === false) {
    echo 'Error while parsing the document';
    exit;
}

$dom_sxe = dom_import_simplexml($sxe);
if (!$dom_sxe) {
    echo 'Error while converting XML';
    exit;
}

$dom = new DOMDocument('1.0');
$dom_sxe = $dom->importNode($dom_sxe, true);
$dom_sxe = $dom->appendChild($dom_sxe);

echo $dom->saveXML();
```

```
// initialize a simplexml object
$sxe = simplexml_load_string('<root/>');

// get a dom interface on the simplexml object
$dom = dom_import_simplexml($sxe);

// dom adds a new element under the root
$element = $dom->appendChild(new DOMElement('dom_element'));
```

```
// dom adds an attribute on the new element
$element->setAttribute('creator', 'dom');

// simplexml adds an attribute on the dom element
$sxe->dom_element['sxe_attribute'] = 'added by simplexml';

// simplexml adds a new element under the root
$element = $sxe->addChild('sxe_element');

// simplexml adds an attribute on the new element
$element['creator'] = 'simplexml';

// dom finds the simplexml element (via DOMNodeList->index)
$element = $dom->getElementsByTagName('sxe_element')->item(0);

// dom adds an attribute on the simplexml element
$element->setAttribute('dom_attribute', 'added by dom');

echo('<pre>');
print_r($sxe);
echo('</pre>');
```

Chapter 121

SimpleXML

SimpleXML 是一个对象到 XML 的映射 API，因此 SimpleXML 对象不是 XML，而是由 SimpleXML 把 XML 元素转换为原生的 PHP 数据类型。

121.1 Overview

SimpleXML 扩展提供了一个非常简单和易于使用的工具集，能将 XML 转换成一个带有一般属性选择器和数组迭代器的对象。

对于 SimpleXML 需要的 XML 字符串，可以将其放入一个文件，或者可以创建一个 XML 文档并使用 simplexml_load_file() 读取它。

Example 135 包含 XML 示例代码的 PHP 文件-example.php

```
<?php
$xmlstr = <<<XML
<?xml version='1.0' standalone='yes'?>
<movies>
 <movie>
  <title>PHP: Behind the Parser</title>
  <characters>
   <character>
    <name>Ms. Coder</name>
    <actor>Onlivia Actora</actor>
   </character>
   <character>
    <name>Mr. Coder</name>
    <actor>El Act&#211;r</actor>
   </character>
```

```
 </characters>
 <plot>
  So, this language. It's like, a programming language. Or is it a
  scripting language? All is revealed in this thrilling horror spoof
  of a documentary.
 </plot>
 <great-lines>
  <line>PHP solves all my web problems</line>
 </great-lines>
 <rating type="thumbs">7</rating>
 <rating type="stars">5</rating>
</movie>
</movies>
XML;
?>
```

Example 136 获取 XML 示例文件中的 <plot> 节点

```
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);
echo $movies->movie[0]->plot;
// 以上例程会输出:
//So, this language. It's like, a programming language. Or is it a
//  scripting language? All is revealed in this thrilling horror spoof
//  of a documentary.
```

访问 XML 文档中包含的 PHP 的命名约定不允许使用的字符（例如连字符）的元素时，可以通过在大括号和引号中封装元素名称来实现。

Example 137 获取 XML 示例文件中的 <line> 节点

```
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);
echo $movies->movie->{'great-lines'}->line;
// 以上例程会输出:
// PHP solves all my web problems
```

Example 138 在 SimpleXML 中访问非唯一元素

```
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);
/* For each <character> node, we echo a separate <name>. */
foreach ($movies->movie->characters->character as $character){
   echo $character->name, ' played by ' . $character->actor, PHP_EOL;
}
// 以上例程会输出:
// Ms. Coder played by Onlivia Actora
// Mr. Coder played by El ActÓr
```

上述示例中的 $movies->movie 属性并不是数组，而是一个可以迭代和可以访问的对象。

除了使用 SimpleXML 来读取元素名称及其值之外，SimpleXML 也可以访问元素属性，以及像访问数组元素一样访问元素的属性。

Example 139　在 SimpleXML 中访问属性

```
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);
/* Access the <rating> nodes of the first movie.
 * Output the rating scale, too. */
foreach ($movies->movie[0]->rating as $rating) {
    switch((string) $rating['type']) { // Get attributes as element indices
    case 'thumbs':
        echo $rating, ' thumbs up';
        break;
    case 'stars':
        echo $rating, ' stars';
        break;
    }
}
// 以上例程会输出:
// 7 thumbs up5 stars
```

如果要对元素或属性与字符串进行比较或将其传递到需要字符串的函数中，必须使用 (string) 将其强制转换为字符串，否则 PHP 将元素视为一个对象。

Example 140　在 SimpleXML 中比较元素和属性与文本

```
include 'example.php';
```

```
$movies = new SimpleXMLElement($xmlstr);
if ((string) $movies->movie->title == 'PHP: Behind the Parser') {
    print 'My favorite movie.';
}

echo htmlentities((string) $movies->movie->title);
// 以上例程会输出:
// My favorite movie.PHP: Behind the Parser
```

两个 SimpleXMLElements 被认为是不同的，即使它们指向 P 同一个元素。

Example 141 在 SimpleXML 中比较两个元素

```
include 'example.php';

$movies1 = new SimpleXMLElement($xmlstr);
$movies2 = new SimpleXMLElement($xmlstr);
var_dump($movies1 == $movies2); // false
// 以上例程会输出:
// bool(false)
```

SimpleXML 内置了对 XPath 的支持，例如下面的示例中使用 XPath 来查找到所有的 <character> 元素:

Example 142 在 SimpleXML 中使用 XPath

```
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);
foreach ($movies->xpath('//character') as $character) {
    echo $character->name, ' played by ', $character->actor, PHP_EOL;
}
// 以上例程会输出:
// Ms. Coder played by Onlivia Actora
// Mr. Coder played by El ActÓr
```

这里的'//'用作通配符，如果要指定绝对路径，需要省略斜杠。
SimpleXML 中的数据不必是常量，而且对象允许操纵其所有元素（例如赋值）。

Example 143 在 SimpleXML 中进行赋值操作

```php
include 'example.php';

$movies = new SimpleXMLElement($xmlstr);

$movies->movie[0]->characters->character[0]->name = 'Miss Coder';

echo $movies->asXML();
```

以上例程的输出如下:

```
<?xml version="1.0" standalone="yes"?>
<movies>
 <movie>
  <title>PHP: Behind the Parser</title>
  <characters>
   <character>
    <name>Miss Coder</name>
    <actor>Onlivia Actora</actor>
   </character>
   <character>
    <name>Mr. Coder</name>
    <actor>El Act&#xD3;r</actor>
   </character>
  </characters>
  <plot>
   So, this language. It's like, a programming language. Or is it a
   scripting language? All is revealed in this thrilling horror spoof
   of a documentary.
  </plot>
  <great-lines>
   <line>PHP solves all my web problems</line>
  </great-lines>
  <rating type="thumbs">7</rating>
  <rating type="stars">5</rating>
 </movie>
</movies>
```

SimpleXML 支持在 XML 中添加元素、子元素和属性等。

Example 144　在 SimpleXML 中新增元素和属性

```php
include 'example.php';
$movies = new SimpleXMLElement($xmlstr);
```

```php
$character = $movies->movie[0]->characters->addChild('character');
$character->addChild('name', 'Mr. Parser');
$character->addChild('actor', 'John Doe');

$rating = $movies->movie[0]->addChild('rating', 'PG');
$rating->addAttribute('type', 'mpaa');

echo $movies->asXML();
```

以上例程的输出如下：

```
<?xml version="1.0" standalone="yes"?>
<movies>
 <movie>
  <title>PHP: Behind the Parser</title>
  <characters>
   <character>
    <name>Ms. Coder</name>
    <actor>Onlivia Actora</actor>
   </character>
   <character>
    <name>Mr. Coder</name>
    <actor>El Act&#xD3;r</actor>
   </character>
  <character><name>Mr. Parser</name><actor>John Doe</actor></character></characters>
  <plot>
   So, this language. It's like, a programming language. Or is it a
   scripting language? All is revealed in this thrilling horror spoof
   of a documentary.
  </plot>
  <great-lines>
   <line>PHP solves all my web problems</line>
  </great-lines>
  <rating type="thumbs">7</rating>
  <rating type="stars">5</rating>
 <rating type="mpaa">PG</rating></movie>
</movies>
```

PHP 提供了在 SimpleXML 和 DOM 格式之间转换 XML 节点的机制，例如下面的示例显示了如何将 DOM 元素更改为 SimpleXML。

Example 145 在 SimpleXML 中实现 DOM 互操作性

```
$dom = new DOMDocument;
$dom->loadXML('<books><book><title>blah</title></book></books>');
if (!$dom) {
    echo 'Error while parsing the document';
    exit;
}

$books = simplexml_import_dom($dom);

echo $books->book[0]->title;
// 以上例程会输出:
// blah
```

libxml 简化了在加载文档时处理 XML 错误的任务，而且 libxml 支持在加载文档时抑制所有 XML 错误，然后迭代错误。

由 libxml_get_errors() 返回的 libXMLError 对象提供的属性可以包括错误的消息（例如行和列（位置））。

Example 146　载入错误的 XML 字符串

```
libxml_use_internal_errors(true);
$sxe = simplexml_load_string("<?xml version='1.0'><broken><xml></broken>");
if ($sxe === false) {
    echo "Failed loading XML\n";
    foreach(libxml_get_errors() as $error) {
        echo "\t", $error->message;
    }
}
// 以上例程会输出:
// Failed loading XML
//    Blank needed here
//    parsing XML declaration: '?>' expected
//    Opening and ending tag mismatch: xml line 1 and broken
//    Premature end of data in tag broken line 1
```

121.1.1 Requirement

SimpleXML 需要 PHP 的 libxml 扩展支持，因此在编译 PHP 时需要指定`--enable-libxml`，尽管这将隐式完成因为 libxml 是缺省开启的。

121.1.2 Resource Type

SimpleXML 函数没有定义资源类型。

121.1.3 Build-in Module

SimpleXML 默认为启用，可以在编译时通过如下选项禁用`--disable-simplexml`。

如果需要手动安装 XML 扩展，可以执行：

```
$ sudo apt-get install php7.0-xml
```

121.1.4 Runtime Configure

SimpleXML 函数没有在 php.ini 中定义配置指令。

121.1.5 Predefined Constants

SimpleXML 没有预先定义常量。

121.2 SimpleXMLElement

SimpleXMLElement 表示 XML 文档中的元素。

```
SimpleXMLElement implements Traversable {
  /* Methods */
  final public __construct ( string $data [, int $options = 0 [, bool $data_is_url =
      false [, string $ns = "" [, bool $is_prefix = false ]]]] )
  public void addAttribute ( string $name [, string $value [, string $namespace ]] )
  public SimpleXMLElement addChild ( string $name [, string $value [, string $namespace
      ]] )
  public mixed asXML ([ string $filename ] )
  public SimpleXMLElement attributes ([ string $ns = NULL [, bool $is_prefix = false ]]
      )
  public SimpleXMLElement children ([ string $ns [, bool $is_prefix = false ]] )
  public int count ( void )
  public array getDocNamespaces ([ bool $recursive = false [, bool $from_root = true ]]
      )
  public string getName ( void )
  public array getNamespaces ([ bool $recursive = false ] )
  public bool registerXPathNamespace ( string $prefix , string $ns )
  public string __toString ( void )
  public array xpath ( string $path )
```

```
}
```

121.2.1 SimpleXMLElement::__construct()

121.2.2 SimpleXMLElement::addAttribute()

121.2.3 SimpleXMLElement::addChild()

```
<?php
$HitourAid='xxx';
$HitourSid='deed';
$timeStamp=time();
$requestType='OAE_FlightSaveOrder';
$signature='yyy';
$userID='';
$productID='';
$corpName=$_SERVER['HTTP_HOST']; // www.hitour.cc
//$corpName=Yii::app()->request->getUserHost(); www.hitour.cc
$sendTicketCity='SHA'; //SHA
$orderDescription='';//玩途订单号
$ticketStatus='A'; //A(立即)
$bookMode='A';//A(携程订位)
$isGroupOrder=false;//false(是否团队票)
$isManualOrder=false;//false(是否手工订单)
$isQuickBooking=false;//false(是否快速预定)
$discountCode='';//折扣码(没有则不填)
$departCityCode='SHA';//SHA(出发城市)
$arriveCityCode='BJS';//BJS(到达城市)
$flightNo='MU5199';//航班号
$subClass='B';//子舱位
$price=1070;//价格(不含基建和燃油)
$takeOffTime='2011-01-11';//出发日期
$productType='Normal';//政策类型
$saleType=2;//销售类型(需要根据指定查询返回值进行替换)
$pid='';//为了获取唯一Productid
$origin_xml = <<<XML
<?xml version="1.0" encoding="utf8"?>
<Request>
    <Header AllianceID="{$HitourAid}" SID="{$HitourSid}" TimeStamp="{$timeStamp}"
        RequestType="{$requestType}" Signature="{$signature}" />
    <FlightSaveOrderRequest>
```

```
    <UserID>{$userID}</UserID>
    <ProductID>{$productID}</ProductID>
    <OrderBaseInfo>
        <CtripCardNo>11111</CtripCardNo>
        <ServiceNo>{$corpName}</ServiceNo>
        <SendTicketCity>{$sendTicketCity}</SendTicketCity>
        <OrderDescription>{$orderDescription}</OrderDescription>
        <TicketStatus>{$ticketStatus}</TicketStatus>
        <BookMode>{$bookMode}</BookMode>
        <IsGroupOrder>{$isGroupOrder}</IsGroupOrder>
        <IsManualOrder>{$isManualOrder}</IsManualOrder>
        <IsQuickBooking>{$isQuickBooking}</IsQuickBooking>
        <DiscountCode/>
    </OrderBaseInfo>
    <FlightSegmentList>
        <FlightSegment>
            <DepartCityCode>{$departCityCode}</DepartCityCode>
            <ArriveCityCode>{$arriveCityCode}</ArriveCityCode>
            <FlightNo>{$flightNo}</FlightNo>
            <SubClass>{$subClass}</SubClass>
            <Price>{$price}</Price>
            <TakeOffTime>{$takeOffTime}</TakeOffTime>
            <ProductType>{$productType}</ProductType>
            <SaleType>{$saleType}</SaleType>
        </FlightSegment>
    </FlightSegmentList>
    <TravelerInfoList>
        <Traveler>
            <PersonName>张三</PersonName>
            <Gender>Male</Gender>
            <ContactInfo>
                <MobilePhone1>13000000000</MobilePhone1>
            </ContactInfo>
            <TravelerCategory>Adult</TravelerCategory>
            <IDCard>
                <IDCardType>ID</IDCardType>
                <IDCardNo>31010519830423281X</IDCardNo>
            </IDCard>
            <AgeType>ADU</AgeType>
            <Birthday>1983-04-23T00:00:00+08:00</Birthday>
            <Nationality>CN</Nationality>
```

```
            <ConfirmStyle>MSG</ConfirmStyle>
            <FFPNoList>
               <FFPNo>
                  <MemberID>456123</MemberID>
                  <Carrier>MU</Carrier>
               </FFPNo>
            </FFPNoList>
         </Traveler>
         <Traveler>
            <PersonName>李四</PersonName>
            <Gender>Female</Gender>
            <ContactInfo>
               <MobilePhone1>13000000001</MobilePhone1>
            </ContactInfo>
            <TravelerCategory>Child</TravelerCategory>
            <IDCard>
               <IDCardType>PASSPORT</IDCardType>
               <IDCardNo>121212</IDCardNo>
            </IDCard>
            <AgeType>CHI</AgeType>
            <Birthday>2008-05-14T00:00:00+08:00</Birthday>
            <Nationality>CN</Nationality>
            <ConfirmStyle>MSG</ConfirmStyle>
            <FFPNoList>
               <FFPNo>
                  <MemberID>456123</MemberID>
                  <Carrier>MU</Carrier>
               </FFPNo>
            </FFPNoList>
         </Traveler>
         <Traveler>
            <PersonName>王五</PersonName>
            <Gender>Male</Gender>
            <TravelerCategory>InfantInLap</TravelerCategory>
            <IDCard>
               <IDCardType>PASSPORT</IDCardType>
               <IDCardNo>212121</IDCardNo>
            </IDCard>
            <AgeType>BAB</AgeType>
            <Birthday>2013-10-31T23:59:59.9999999+08:00</Birthday>
            <Nationality>CN</Nationality>
```

```
            <ConfirmStyle>TEL</ConfirmStyle>
            <FFPNoList>
                <FFPNo>
                    <MemberID>456123</MemberID>
                    <Carrier>MU</Carrier>
                </FFPNo>
            </FFPNoList>
        </Traveler>
    </TravelerInfoList>
    <ContactInfo>
        <PersonName>张三</PersonName>
        <Gender>Male</Gender>
        <ContactInfo>
            <ContactTel>021-34064880-12341</ContactTel>
            <ContactFax>021-34064880-12342</ContactFax>
            <ContactEmail>test@ctrip.com</ContactEmail>
            <MobilePhone1>13000000000</MobilePhone1>
            <MobilePhone2>13100000000</MobilePhone2>
        </ContactInfo>
        <ConfirmStyle>MSG</ConfirmStyle>
    </ContactInfo>
    <DeliveryInfo>
        <DeliveryType>PJS</DeliveryType>
        <FlightAgency>0</FlightAgency>
        <TicketType>NA</TicketType>
        <ThirdPartyDelivery>
            <Province>上海</Province>
            <City>上海</City>
            <District>长宁区</District>
            <Addr>福泉路 99 号</Addr>
            <PostCode>200335</PostCode>
            <ReceiverName>张三</ReceiverName>
            <ExpressPhone>13000000000</ExpressPhone>
            <ExpressInfo>
                <MailingType>EP</MailingType>
                <MailingPayType>Normal</MailingPayType>
                <IntegralAmount>0</IntegralAmount>
                <SendTicketFee>0</SendTicketFee>
            </ExpressInfo>
        </ThirdPartyDelivery>
    </DeliveryInfo>
```

```
        <OrderMiscInfo>
            <IsConnectedSubOrder>true</IsConnectedSubOrder>
            <CashBack>0</CashBack>
        </OrderMiscInfo>
        <Action>CreateOrder</Action>
    </FlightSaveOrderRequest>
</Request>
XML;

$requestXMLBody=new SimpleXMLElement($origin_xml);
//如果有回程则传入回程FlightSegment,没有回程则不传
$returnFlightSegment = <<<XML
<FlightSegment>
<DepartCityCode>{$departCityCode}</DepartCityCode>
<ArriveCityCode>{$arriveCityCode}</ArriveCityCode>
<FlightNo>{$flightNo}</FlightNo>
<SubClass>{$subClass}</SubClass>
<Price>{$price}</Price>
<TakeOffTime>{$takeOffTime}</TakeOffTime>
<ProductType>{$productType}</ProductType>
<SaleType>{$saleType}</SaleType>
</FlightSegment>
XML;
$returnFlightSegmentXML=new SimpleXMLElement($returnFlightSegment);
$requestXMLBody->FlightSaveOrderRequest[0]->FlightSegmentList[0]->addChild('
    FlightSegment');
foreach($returnFlightSegmentXML as $k=>$value){
    $requestXMLBody->FlightSaveOrderRequest[0]->FlightSegmentList[0]->FlightSegment[1]->
        addChild($k,$value);
}

echo $requestXMLBody->asXML();
```

121.2.4 SimpleXMLElement::asXML()

121.2.5 SimpleXMLElement::attributes

121.2.6 SimpleXMLElement::children()

121.2.7 SimpleXMLElement::count()

121.2.8 SimpleXMLElement::getDocNamespaces()

121.2.9 SimpleXMLElement::getName()

121.2.10 SimpleXMLElement::getNamespaces()

121.2.11 SimpleXMLElement::registerXPathNamespace()

121.2.12 SimpleXMLElement::saveXML()

121.2.13 SimpleXMLElement::__toString()

121.2.14 SimpleXMLElement::xpath()

121.3 SimpleXMLIterator

SimpleXMLIterator 对 SimpleXMLElement 对象的所有节点提供递归迭代。

```
SimpleXMLIterator extends SimpleXMLElement implements RecursiveIterator , Countable {
  /* Methods */
  public mixed current ( void )
  public SimpleXMLIterator getChildren ( void )
  public bool hasChildren ( void )
  public mixed key ( void )
  public void next ( void )
  public void rewind ( void )
  public bool valid ( void )
  /* Inherited methods */
  final public SimpleXMLElement::__construct ( string $data [, int $options = 0 [, bool
      $data_is_url = false [, string $ns = "" [, bool $is_prefix = false ]]]] )
  public void SimpleXMLElement::addAttribute ( string $name [, string $value [, string
      $namespace ]] )
  public SimpleXMLElement SimpleXMLElement::addChild ( string $name [, string $value [,
      string $namespace ]] )
  public mixed SimpleXMLElement::asXML ([ string $filename ] )
```

```
  public SimpleXMLElement SimpleXMLElement::attributes ([ string $ns = NULL [, bool
      $is_prefix = false ]] )
  public SimpleXMLElement SimpleXMLElement::children ([ string $ns [, bool $is_prefix =
      false ]] )
  public int SimpleXMLElement::count ( void )
  public array SimpleXMLElement::getDocNamespaces ([ bool $recursive = false [, bool
      $from_root = true ]] )
  public string SimpleXMLElement::getName ( void )
  public array SimpleXMLElement::getNamespaces ([ bool $recursive = false ] )
  public bool SimpleXMLElement::registerXPathNamespace ( string $prefix , string $ns )
  public string SimpleXMLElement::__toString ( void )
  public array SimpleXMLElement::xpath ( string $path )
}
```

121.3.1 SimpleXMLIterator::current()

121.3.2 SimpleXMLIterator::getChildren()

121.3.3 SimpleXMLIterator::hasChildren()

121.3.4 SimpleXMLIterator::key()

121.3.5 SimpleXMLIterator::next()

121.3.6 SimpleXMLIterator::rewind()

121.3.7 SimpleXMLIterator::valid()

121.4 SimpleXML Functions

121.4.1 simplexml_import_dom()

121.4.2 simplexml_load_file()

121.4.3 simplexml_load_string()

Part XXV

JSON

Chapter 122

Overview

Chapter 123

JSON

PHP 的 json 扩展实现了对 JSON（JavaScript Object Notation）的数据转换，其中解码分析器基于 Douglas Crockford 的 JSON_checker。

PHP 实现了 JSON 的 RFC 7159 标准中指定的 JSON 规范的超集。

JSON 扩展是 PHP 核心的一部分，而且该扩展没有在 php.ini 中定义配置指令，也没有定义自己的资源类型。

JSON 扩展引入了如下的预定义常量，并且仅在此扩展编译入 PHP 或在运行时动态载入时可用。

表 123.1: json_last_error() 返回的错误类型

常量名字	常量值	说明
JSON_ERROR_NONE （integer）	0	没有错误发生。
JSON_ERROR_DEPTH（integer）	1	到达了最大堆栈深度。
JSON_ERROR_STATE_MISMATCH (integer)	2	出现了下溢（underflow）或者模式不匹配。
JSON_ERROR_CTRL_CHAR (integer)	3	控制字符错误，可能是编码不对。
JSON_ERROR_SYNTAX（integer）	4	语法错误。

常量名字	常量值	说明
JSON_ERROR_UTF8 (integer)	5	异常的 UTF-8 字符，也许是因为不正确的编码。
JSON_ERROR_RECURSION (integer)	6	The object or array passed to json_encode() include recursive references and cannot be encoded. If the JSON_PARTIAL_OUTPUT_ON_ERROR option was given, NULL will be encoded in the place of the recursive reference.
JSON_ERROR_INF_OR_NAN (integer)	7	The value passed to json_encode() includes either NAN or INF. If the JSON_PARTIAL_OUTPUT_ON_ERROR option was given, 0 will be encoded in the place of these special numbers.
JSON_ERROR_UNSUPPORTED_TYPE (integer)	8	A value of an unsupported type was given to json_encode(), such as a resource. If the JSON_PARTIAL_OUTPUT_ON_ERROR option was given, NULL will be encoded in the place of the unsupported value.
JSON_ERROR_INVALID_PROPERTY_NAME	9	
JSON_ERROR_UTF16	19	

表 123.2: json_encode() 的 form 选项常量

常量名字	常量值	说明
JSON_HEX_TAG (integer)	1	所有的 < 和 > 转换成 \u003C 和 \u003E。

常量名字	常量值	说明
JSON_HEX_AMP (integer)	2	所有的 & 转换成 \u0026。
JSON_HEX_APOS (integer)	4	所有的' 转换成 \u0027。
JSON_HEX_QUOT (integer)	8	所有的" 转换成 \u0022。
JSON_FORCE_OBJECT (integer)	16	使一个非关联数组输出一个类（Object）而非数组。在数组为空而接受者需要一个类（Object）的时候尤其有用。
JSON_NUMERIC_CHECK (integer)	32	将所有数字字符串编码成数字（numbers）。
JSON_UNESCAPED_SLASHES (integer)	64	不要编码 /。
JSON_PRETTY_PRINT (integer)	128	用空白字符格式化返回的数据。
JSON_UNESCAPED_UNICODE (integer)	256	以字面编码多字节 Unicode 字符（默认是编码成 \uXXXX）。
JSON_PARTIAL_OUTPUT_ON_ERROR (integer)	512	
JSON_PRESERVE_ZERO_FRACTION (integer)	1024	
JSON_BIGINT_AS_STRING (integer)	2	将大数字编码成原始字符原来的值。

123.1 json_decode()

对 JSON 格式的字符串进行解码

123.2 json_encode()

对变量进行 JSON 编码

123.3 json_last_error_msg()

Returns the error string of the last json_encode() or json_decode() call

123.4 json_last_error()

返回最后发生的错误

Chapter 124

JSONP

JSONP 将 JSON 数据包装在一个回调函数中。

```
callback({/*.. JSON data .. */});
```

具体来说，JSONP（JSON with Padding）是 JSON 的一种“使用模式”，可以让网页从别的网域请求数据，另一个解决这个问题的新方法是跨来源资源共享。

由于浏览器的同源策略的限制，一般来说位于 server1.example.com 的网页无法与不是 server1.example.com 的服务器沟通，而 HTML 的 <script> 元素是一个例外。

利用 <script> 元素的这个开放策略，网页可以得到从其他来源动态产生的 JSON 数据，而这种使用模式就是所谓的 JSONP。用 JSONP 抓到的数据并不是 JSON，而是任意的 JavaScript，用 JavaScript 解释器运行而不是用 JSON 解析器解析。

为了理解这种模式的原理，首先想象有一个回传 JSON 文件的 URL，而 JavaScript 程序可以用 XMLHttpRequest 向这个 URL 请求数据。假设我们的 URL 是http://server2.example.com/RetrieveUser?UserId=xxx，如果 UserId 是 1823，且当浏览器通过 URL 传输 UserId，也就是抓取http://server2.example.com/RetrieveUser?UserId=1823，得到：

```
{"Name": "小明", "Id" : 1823, "Rank": 7}
```

上述这个 JSON 数据可能是依据传过去 URL 的查询参数动态产生的。

这个时候，把 <script> 元素的 src 属性设成一个回传 JSON 的 URL 是可以想像的，这也代表从 HTML 页面通过 script 元素抓取 JSON 是可能的。

然而，一份 JSON 文件并不是一个 JavaScript 程序。为了让浏览器可以在 <script> 元素运行，从 src 里 URL 回传的必须是可运行的 JavaScript。在 JSONP 的使用模式里，该 URL 回传的是由函数调用包起来的动态生成 JSON，这就是 JSONP 的“填充（padding）”或是“前缀（prefix）”的由来。

惯例上浏览器提供回调函数的名称当作送至服务器的请求中命名查询参数的一部分，例如：

```
<script type="text/javascript" src="http://server2.example.com/RetrieveUser?UserId=1823&
    jsonp=parseResponse">
</script>
```

服务器会在传给浏览器前将 JSON 数据填充到回调函数（parseResponse）中。浏览器得到的回应已不是单纯的数据叙述而是一个脚本。在本例中，浏览器得到的是：

*/,

```
parseResponse({"Name": "小明", "Id" : 1823, "Rank": 7})
```

虽然这个填充（前缀）“通常”是浏览器运行背景中定义的某个回调函数，它也可以是变量赋值、if 叙述或者是其他 JavaScript 叙述。JSONP 要求（也就是使用 JSONP 模式的请求）的回应不是 JSON 也不被当作 JSON 解析——回传内容可以是任意的表达式，甚至不需要有任何的 JSON，不过惯例上填充部分还是会触发函数调用的一小段 JavaScript 片段，而这个函数调用是作用在 JSON 格式的数据上的。

另一种说法—典型的 JSONP 就是把既有的 JSON API 用函数调用包起来以达到跨域访问的解决方案。

为了要引导一个 JSONP 调用（或者说，使用这个模式），你需要一个 script 元素。因此，浏览器必须为每一个 JSONP 要求加（或是重用）一个新的、有所需 src 值的 <script> 元素到 HTML DOM 里—或者说是“注入”这个元素。浏览器运行该元素，抓取 src 里的 URL，并运行回传的 JavaScript。

上述的原理使得 JSONP 被称作是一种“让用户利用 script 元素注入的方式绕开同源策略”的方法。

使用远程网站的 script 标签会让远程网站得以注入任何的内容至网站里。如果远程的网站有 JavaScript 注入漏洞，原来的网站也会受到影响。

现在有一个正在进行项目在定义所谓的 JSON-P 严格安全子集，使浏览器可以对 MIME 类别是“application/json-p”请求做强制处理。如果回应不能被解析为严格的 JSON-P，浏览器可以丢出一个错误或忽略整个回应。

粗糙的 JSONP 部署很容易受到跨站请求伪造（CSRF/XSRF）的攻击。因为 HTML <script> 标签在浏览器里不遵守同源策略，恶意网页可以要求并获取属于其他网站的 JSON 数据。当用户正登录那个其他网站时，上述状况使得该恶意网站得以在恶意网站的环境下操作该 JSON 数据，可能泄漏用户的密码或是其他敏感数据。

只有在该 JSON 数据含有不该泄漏给第三方的隐密数据，且服务器仅靠浏览器的同源策略阻挡不正常要求的时候这才会是问题。若服务器自己决定要求的专有性，并只在要求正常的情况下输出数据则没有问题。只靠 Cookie 并不足以决定要求是合法的，这很容易受到跨站请求伪造攻击。

乔治 · 詹姆提（George Jempty）在 2005 年夏天建议在 JSON 前面选择性的加上变量赋值，鲍勃 · 伊波利托（Bob Ippolito）于 2005 年 12 月提出了 JSONP 最原始的提案，其中填充

部分已经是回调函数，现在已经有很多 Web 2.0 应用程序开始使用 JSONP 提案，例如 Dojo Toolkit、Google Web Toolki 与 Web 服务等。

```
<?php
header("content-type: text/javascript");

if(isset($_GET['name']) && isset($_GET['callback'])) {
    $obj->name = $_GET['name'];
    $obj->message = "Hello " . $obj->name;

    echo $_GET['callback']. '(' . json_encode($obj) . ');';
}
```

下面的示例说明如何使用 jQuery 来请求 JSONP 数据：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>JQuery JSONP</title>
    <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.2/jquery.min.js"></
        script>
    <script>
    $(document).ready(function(){

        $("#useJSONP").click(function(){
            $.ajax({
                url: 'http://domain.com/jsonp-demo.php',
                data: {name: 'Chad'},
                dataType: 'jsonp',
                jsonp: 'callback',
                jsonpCallback: 'jsonpCallback',
                success: function(){
                    alert("success");
                }
            });
        });

    });

    function jsonpCallback(data){
        $('#jsonpResult').text(data.message);
    }
    </script>
```

```
  </head>

  <body>
    <input type="button" id="useJSONP" value="Use JSONP"></input><br /><br />
    <span id="jsonpResult"></span>
  </body>
</html>
```

为了使用 jQuery 发送 JSONP 数据，需要设置 dataType 为 jsonp，并且为 jsonp 设置回调函数，因此请求 JSONP 数据的的 URL 类似于http://domain.com/jsonp-demo.php?callback=jsonpCallback&name=Chad。

对应的对 PHP 的请求产生的结果如下：

```
jsonpCallback({"name":"Chad","message":"Hello Chad"});
```

为了严格防止跨站请求伪造攻击，需要在 PHP 返回的结果的返回头中设置：

```
header("content-type: Access-Control-Allow-Origin: *");
header("content-type: Access-Control-Allow-Methods: GET");
```

如果使用托管服务或不能完全控制服务器端逻辑，可能服务器主机会阻止 JSONP 请求，因此需要在 Web 服务器（例如 Apache）进行如下配置：

```
Header set Access-Control-Allow-Origin "mydomain.com"
```

除了可以使用字符串函数移除 JSON 数据包头之外，也可以定义 jsonp_decode() 和 jsonp_encode() 来处理 JSONP。

124.1 jsonp_decode()

```
/**
 * Decodes a JSONP string and falls back to JSON decoding in case
 * the value passed is not JSONP.
 *
 * Provides the same interface as the built-in json_decode function
 *
 * @param string $jsonp
 * @param bool $assoc
 *
 * @return mixed
 */
function jsonp_decode($jsonp, $assoc = false) {
   // PHP 5.3 adds depth as third parameter to json_decode
```

```
    $jsonLiterals = array('true' => 1, 'false' => 1, 'null');
    if(preg_match('/^[^{"\d]/', $jsonp) && !isset($jsonLiterals[$jsonp])) { // we have
        JSONP
      $jsonp = substr($jsonp, strpos($jsonp, '('));
    }
    return json_decode(trim($jsonp,'();'), $assoc);
}
```

124.2 jsonp_encode()

```
/**
 * Encodes the value as JSONP with the given function name.
 *
 * @param mixed $value
 * @param string $function_name
 *
 * @return string
 */
function jsonp_encode($value, $function_name = 'cb') {
    // 5.3 adds options as second parameter to json_encode
    return sprintf('%s(%s);', $function_name, json_encode($value));;
}
```

Chapter 125

JSONSerializable

实现 JsonSerializable 的类可以在 json_encode() 时定制自己的 JSON 表示法。

```
JsonSerializable {
    /* 方法 */
    abstract public mixed jsonSerialize ( void )
}
```

125.1 JSONSerializable::jsonSerialize()

指定需要被序列化成 JSON 的数据

```
abstract public mixed JsonSerializable::jsonSerialize ( void )
```

把对象（Object）序列化成能被 json_encode() 原生地序列化的值，因此返回值是能被 json_encode() 序列化的数据，这个值可以是除了 resource 外的任意类型。

Example 147 使用 JSONSerializable::jsonSerialize() 返回数组

```
<?php
class ArrayValue implements JsonSerializable {
    public function __construct(array $array) {
        $this->array = $array;
    }

    public function jsonSerialize() {
        return $this->array;
    }
}
```

```
$array = [1, 2, 3];
echo json_encode(new ArrayValue($array), JSON_PRETTY_PRINT);
// 以上例程会输出:
[
    1,
    2,
    3
]
```

Example 148 使用 JSONSerializable::jsonSerialize() 返回关联数组

```
<?php
class ArrayValue implements JsonSerializable {
    public function __construct(array $array) {
        $this->array = $array;
    }

    public function jsonSerialize() {
        return $this->array;
    }
}

$array = ['foo' => 'bar', 'quux' => 'baz'];
echo json_encode(new ArrayValue($array), JSON_PRETTY_PRINT);
// 以上例程会输出:
{
    "foo": "bar",
    "quux": "baz"
}
```

Example 149 使用 JSONSerializable::jsonSerialize() 返回整型值

```
<?php
class IntegerValue implements JsonSerializable {
    public function __construct($number) {
        $this->number = (integer) $number;
    }

    public function jsonSerialize() {
        return $this->number;
    }
}
```

```
echo json_encode(new IntegerValue(1), JSON_PRETTY_PRINT);
// 以上例程会输出:
1
```

Example 150 使用 JSONSerializable::jsonSerialize() 返回字符串值

```
<?php
class StringValue implements JsonSerializable {
    public function __construct($string) {
        $this->string = (string) $string;
    }

    public function jsonSerialize() {
        return $this->string;
    }
}

echo json_encode(new StringValue('Hello!'), JSON_PRETTY_PRINT);
// 以上例程会输出:
"Hello!"
```

Part XXVI

Java

Chapter 126

Overview

除了可以使用 Zend Server 提供的 Java Bridge 功能之外，还可以使用开源的 PHP Java Bridge

安装基础安装包（例如 php5-dev，sun java 和 automake 等）

```
# apt-get install build-essential php5-dev sun-java6-jre \
   sun-java6-jdk sun-java6-source automake
```

下载 PHP/Java Bridge 源代码

```
# cd /usr/src
# wget http://internap.dl.sourceforge.net/sourceforge/php-java-bridge/php-java-bridge_5
    .2.0.tar.gz
# tar xzfv php-java-bridge_5.2.0.tar.gz
```

安装 PHP/Java Bridge

```
# cd /usr/src/php-java-bridge-5.2.0
# phpize
# ./configure --with-java=/usr/lib/jvm/java-6-sun-1.6.0.03,/usr/lib/jvm/java-6-sun
    -1.6.0.03
# make
# chmod +x install.sh
# ./install.sh
```

拷贝相关 *.inc 到对应的目录下

```
# mkdir /usr/share/php/java
# cp *.inc /usr/share/php/java
```

禁用 java.hosts 和 java.servlet 并执行测试：

```
# php test.php
```

默认情况下，Java 仅仅在 PHP 的命令行模式下可用，如果需要在全局使用 Java，那么可以在 php.ini 修改如下配置：

```
;; java.ini: Activate the PHP/Java extension

;; zend_extension = "/path/to/java.so"
extension = java.so

;; If you have installed the java-servlet.ini leave this file alone,
;; edit the servlet or standalone ini file. Otherwise uncomment the
;; following java section and one of the following options:
[java]
```

Zend 的 Java Bridge 已经有一个名为 java_require 的函数，开源的 PHP/Java Bridge 需要包括这个文件，而且 PHP/Java Bridge 需要 java_cast() 函数，Zend 的 Java Bridge 不需要这个文件和函数。

根据上述两个差异，开发者需要在 bootstrap 文件（或对应代码）覆盖它们，这样应用程序就可以与这两个 Java/Bridge 一起工作。

```
/* no java_require() include the java.inc for PHP/Java Bridge */
if ( ! function_exists('java_require') ) {
   include "java/Java.inc";
}

/* declare this, it doesn't exist with Zend Java, but is needed for PHP/Java Bridge */
if ( ! function_exists('java_cast')) {
   function java_cast($whatever) {
      return $whatever;
   }
}
```

上述解决并非完美，在实际情况中需要根据需求进行相应的配置。

```
// Get a reference to a container-managed resource from WebLogic Server
function getWebLogicResource($jndiName) {
  // Set WebLogic-specific parameters for creation of jndi naming context:
  // (weblogic.jar must be on the PHP-Java Bridge classpath)
  $envt = array(
     "java.naming.factory.initial" => "weblogic.jndi.WLInitialContextFactory",
     "java.naming.provider.url" => "t3://localhost:7001" // replace with your target
         server URL
  );
```

```
  // The bridge creates the naming context object in the local JVM.
  $ctx = new Java('javax.naming.InitialContext', $envt);

  // Now JNDI crosses the network, to obtain the resource.
  return $ctx->lookup($jndiName);
}
```

Part XXVII

XML-RPC

Chapter 127

Overview

127.1 XML-RPC Extension

127.2 XML-RPC Library

127.3 XML-RPC.NET

XML-RPC.NET 是一个使用.NET 技术环境（支持.NET 2.0 及以上）实现的客户端和服务器端代码库

- 基于接口的 XML-RPC 服务器和客户端定义
- 类型安全客户端代理的代码生成
- Silverlight 和 Windows Phone 7 支持
- 支持 XML-RPC 和 SOAP 的 ASP.NET Web Service
- 客户端支持异步调用
- 客户端支持各种 XML 编码和 XML 缩进样式（一些其他 XML-RPC 服务器实现只接受某些缩进样式）
- 内置对服务器上 XML-RPC Introspection API 的支持
- 根据 XML-RPC end-point 的 URL 动态生成文档页面
- 支持将 XML-RPC 方法和 struct 成员名称映射到与.NET 兼容的名称
- 支持客户端和服务器中的 Unicode XML-RPC 字符串
- 在.NET 和 XML-RPC 类型之间映射时支持可选的 struct 成员
- 支持 XML-RPC 的 <i8> 和 <nil> 扩展
- 在客户端和服务器上支持.NET Remoting

127.3.1 XML-RPC Clients

用户使用 XML-RPC.NET 定义一个 XML-RPC 接口并使用 XMLRpcProxyGen 类来自动生成代码。

```
[XmlRpcUrl("http://www.cookcomputing.com/xmlrpcsamples/RPC2.ashx")]
public interface IStateName : IXmlRpcProxy
{
    [XmlRpcMethod("examples.getStateName")]
    string GetStateName(int stateNumber);
}
```

XMLRpcProxyGen 类的静态方法生成 XMLRpcProxy 实例。

```
IStateName proxy = XmlRpcProxyGen.Create<IStateName>();
```

XMLRpcProxy 实例的方法可以创建 XML-RPC 请求来调用远程 XML-RPC 服务器。

```
string stateName = proxy.GetStateName(41);
```

127.3.2 XML-RPC Services

XML-RPC.NET 将 XML-RPC 服务实现为在 Microsoft IIS Web 服务器环境中运行的服务。

XML-RPC 服务的模型是作为 ASP.Net 的一部分实现的基于 SOAP 的 Web 服务。通过创建从 XmlRpcService 基类派生的类以及通过带有 XmlRpcMethod 属性的 XML-RPC 装饰要公开的方法，实现 XML-RPC 服务（在任何 CLS 兼容的语言中，例如 C ＃，VB.Net 等）。

```
public class StateNameService : XmlRpcService
{
  [XmlRpcMethod("examples.getStateName",
    Description="Return name of state given its number")]
  public string getStateName(int stateNum)
  {
    if (stateNum == 41)
      return "South Dakota";
    else
      return "Don't know";
  }
}
```

除了指定 XML-RPC 方法之外，XmlRpcMethod 属性还用于指定该方法使用 XML-RPC 协议作为“examples.getStateName”调用，而不是在 Service 类中使用的方法的名称。当分配给描述的字符串用于通过 HTTP GET 请求调用服务时如下所述的自动文档生成。

或者，服务类也可以从定义 XML-RPC 方法的接口派生，而且接口然后还可以用于生成如上所述的代理类。

```
public interface IStateName
{
    [XmlRpcMethod("examples.getStateName")]
    string GetStateName(int stateNumber);
}

public interface IStateNameProxy : IStateName, IXmlRpcProxy
{
}

public class StateNameService : XmlRpcService, IStateName
{
  public string getStateName(int stateNum)
  {
    if (stateNum == 41)
        return "South Dakota";
    else
      return "Don't know";
  }
}
```

类可以实现许多 XML-RPC 方法，而不仅仅是这些示例中的单个方法。

将生成的程序集 DLL 放置在 IIS 虚拟目录的 bin 目录中，并使用 web.config 文件将 HTTP 请求分发到由 XmlRpcService 类实现的自定义处理程序。例如，如果 cookcomputing.com 有一个名为 xmlrpc 的虚拟目录，并且以下配置文件放在 xmlrpc 的根目录中：

```
<configuration>
  <system.web>
    <httpHandlers>
      <add verb="*" path="statename.rem"
        type="CookComputing.StateNameService, StateNameService" />
    </httpHandlers>
  </system.web>
</configuration>
```

上述示例中的 XML-RPC Serive 提供的功能可以通过类似http://localhost/xmlrpc/statename.rem的 XML-RPC 请求来调用。

注意，类型是装配限定的，类的名称是在 AssemblyName StateService（即 StateNameService.dll）中的 CookComputing.StateNameService。

如果调用者不通过 HTTP POST 请求发出 XML-RPC 请求，而是向同一个 URL 发出 HTTP GET 请求，则服务会返回一个自动生成的描述其自身的页面。对于上面的示例服务，返回此页面。

或者, 服务可以通过 XMLRPC.NET 的.NET Remoting 支持来实现, 然后可以由 XML-RPC 或 SOAP 协议访问相同的服务。

Chapter 128

XML-RPC Extension

XML-RPC 提供的函数可以用来编写 XML-RPC 的服务器和客户端代码。

XML-RPC 扩展需要 libxml 扩展到的支持，不过 libxml 默认是开启的。

默认情况下，在 PHP 中是不能使用 XML-RPC 支持的，需要使用`--with-xmlrpc[=DIR]`配置选项编译 PHP 才能够使用 XML-RPC 支持。

128.1 Runtime Configuration

XML-RPC 扩展没有在 php.ini 中定义配置指令。

128.2 Resource Type

XML-RPC 扩展通过 xmlrpc_server_create() 定义了一个 XML-RPC 服务器数据类型。

128.3 Predefined Constant

XML-RPC 扩展没有定义常量。

128.4 XML-RPC Client

下面是一个 XML-RPC PHP Client 的示例，在 HTML 代码中嵌入了 XML-RPC 调用。

```
<html>
<head>
<title>XML-RPC PHP Demo</title>
</head>
```

```
<body>
<h1>XML-RPC PHP Demo</h1>

<?php
include 'xmlrpc.inc';

// Make an object to represent our server.
$server = new xmlrpc_client('/api/sample.php',
                        'xmlrpc-c.sourceforge.net', 80);

// Send a message to the server.
$message = new xmlrpcmsg('sample.sumAndDifference',
                      array(new xmlrpcval(5, 'int'),
                            new xmlrpcval(3, 'int')));
$result = $server->send($message);

// Process the response.
if (!$result) {
    print "<p>Could not connect to HTTP server.</p>";
} elseif ($result->faultCode()) {
    print "<p>XML-RPC Fault #" . $result->faultCode() . ": " .
        $result->faultString();
} else {
    $struct = $result->value();
    $sumval = $struct->structmem('sum');
    $sum = $sumval->scalarval();
    $differenceval = $struct->structmem('difference');
    $difference = $differenceval->scalarval();
    print "<p>Sum: " . htmlentities($sum) .
        ", Difference: " . htmlentities($difference) . "</p>";
}
?>
</body></html>
```

128.5 XML-RPC Server

下面的示例使用 PHP 实现了一个 XML-RPC 服务器，远程客户端可以通过类似http://localhost/path/sumAndDifference.php的方式来调用 XML-RPC Web 服务。

```
<?php
include 'xmlrpc.inc';
```

```
include 'xmlrpcs.inc';

function sumAndDifference ($params) {

   // Parse our parameters.
   $xval = $params->getParam(0);
   $x = $xval->scalarval();
   $yval = $params->getParam(1);
   $y = $yval->scalarval();

   // Build our response.
   $struct = array('sum' => new xmlrpcval($x + $y, 'int'),
                'difference' => new xmlrpcval($x - $y, 'int'));
   return new xmlrpcresp(new xmlrpcval($struct, 'struct'));
}

// Declare our signature and provide some documentation.
// (The PHP server supports remote introspection. Nifty!)
$sumAndDifference_sig = array(array('struct', 'int', 'int'));
$sumAndDifference_doc = 'Add and subtract two numbers';

new xmlrpc_server(array('sample.sumAndDifference' =>
                    array('function' => 'sumAndDifference',
                         'signature' => $sumAndDifference_sig,
                         'docstring' => $sumAndDifference_doc)));
```

128.6 Functions

128.6.1 xmlrpc_decode_request()

将 XML 译码为 PHP 本身的类型

128.6.2 xmlrpc_decode()

将 XML 译码为 PHP 本身的类型

128.6.3 xmlrpc_encode_request()

为 PHP 的值生成 XML

128.6.4 xmlrpc_encode()

为 PHP 的值生成 XML

128.6.5 xmlrpc_get_type()

为 PHP 的值获取 xmlrpc 的类型

128.6.6 xmlrpc_is_fault()

Determines if an array value represents an XMLRPC fault

128.6.7 xmlrpc_parse_method_description()

将 XML 译码成方法描述的列表

128.6.8 xmlrpc_server_add_introspection_data()

添加自我描述的文档

128.6.9 xmlrpc_server_call_method()

解析 XML 请求同时调用方法

128.6.10 xmlrpc_server_create()

创建一个 xmlrpc 服务端

128.6.11 xmlrpc_server_destroy()

销毁服务端资源

128.6.12 xmlrpc_server_register_introspection_callback()

注册一个 PHP 函数用于生成文档

128.6.13 xmlrpc_server_register_method()

注册一个 PHP 函数用于匹配 xmlrpc 方法名

128.6.14 xmlrpc_set_type

为一个 PHP 字符串值设置 xmlrpc 的类型、base64 或日期时间

Chapter 129

XML-RPC Library

XML-RPC 库（https://github.com/gggeek/phpxmlrpc）是一个实现了 XML-RPC 协议的 PHP 库，可以用来构建 XML-RPC 客户端和 XML-RPC 服务器。

XML-RPC 库不需要 PHP 内置的 xmlrpc 扩展，即使已经安装了 xmlrpc 扩展，也不会影响到 XML-RPC 库的使用，推荐使用 Composer 来安装 phpxmlrpc 库，例如：

```
$ composer require phpxmlrpc/phpxmlrpc:4.0
```

或者，可以在 composer.json 中添加如下依赖：

*/,

```
"require": {
    ...,
    "phpxmlrpc/phpxmlrpc": "~4.0"
},
```

如果不使用 Composer 来管理依赖，可以在 src/Atuloader.php 中找到该库的自动加载器，设置自动加载器的示例代码如下：

```
<?php
// File example: src/script.php

// update this to the path to the "vendor/" directory, relative to this file
require_once __DIR__.'/../vendor/autoload.php';

use PhpXmlRpc\Value;
use PhpXmlRpc\Request;
use PhpXmlRpc\Client;

$client = new Client('http://some/server');
$response = $client->send(new Request('method', array(new Value('parameter'))));
```

XML-RPC 的 4.0 版本相比 3.0 版本的改动包括所有类都使用命名空间，并遵循 CamelCase 命名约定，现有的类方法和成员被保留，所有新方法名称遵循 camelCase。

129.1 xmlrpc_client

129.2 PhpXmlRpc\Client

129.3 xmlrpc_server

129.4 PhpXmlRpc\Server

129.5 xmlrpcmsg

129.6 PhpXmlRpc\Request

129.7 xmlrpcresp

129.8 PhpXmlRpc\Response

129.9 xmlrpcval

129.10 PhpXmlRpc\Value

Chapter 130

XML-RPC.NET

Part XXVIII

Miscellaneous

Chapter 131

Overview

131.1 Requirement

杂项函数不需要其他扩展支持。

131.2 Resource Type

杂项函数没有定义资源类型。

131.3 Build-in Module

杂项函数是 PHP 核心的一部分，由 PHP 核心直接支持。

131.4 Runtime Configure

杂项函数的行为受到 php.ini 中的配置项的影响。

表 131.1: Network Configuration 选项

名字	默认	可修改范围	说明
ignore_user_abort	"0"	PHP_INI_ALL	
highlight.string	"#DD0000"	PHP_INI_ALL	
highlight.comment	"#FF8000"	PHP_INI_ALL	

名字	默认	可修改范围	说明
highlight.keyword	"#007700"	PHP_INI_ALL	
highlight.bg	"#FFFFFF"	PHP_INI_ALL	已移除
highlight.html	"#0000BB"	PHP_INI_ALL	
highlight.html	"#000000"	PHP_INI_ALL	
browscap	NULL	PHP_INI_SYSTEM	

131.4.1 ignore_user_abort

与 ignore_user_abort() 相关（默认值为 FALSE），如果设置为 TRUE ，在客户端断开连接后，脚本不会被中止。

131.4.2 highlight.bg

131.4.3 highlight.comment

131.4.4 highlight.default

131.4.5 highlight.html

131.4.6 highlight.keyword

131.4.7 highlight.string

语法高亮的颜色，可设置为 `<font color="??????">` 中任何可接受的代码。

131.4.8 browscap

与 get_browser() 相关，设置浏览器功能文件的位置和文件名 (例如 browscap.ini)。

131.5 Predefined Constants

杂项函数预先定义了一些常量，并且这些常量作为 PHP 核心的一部分或在运行时总是可用的。

表 131.2: 杂项函数的常量

常量名	说明
CONNECTION_ABORTED (integer)	
CONNECTION_NORMAL (integer)	
CONNECTION_TIMEOUT (integer)	
__COMPILER_HALT_OFFSET__ (integer)	

Chapter 132

Functions

132.1 connection_aborted()

检查客户端是否已经断开

132.2 connection_status()

返回连接的状态位

132.3 constant()

返回一个常量的值

132.4 define()

定义一个常量

132.5 defined()

检查某个名称的常量是否存在

132.6 die()

等同于 exit()

132.7 eval()

把字符串作为 PHP 代码执行

132.8 exit()

输出一个消息并且退出当前脚本

132.9 get_browser()

获取浏览器具有的功能

132.10 __halt_compiler()

中断编译器的执行

132.11 highlight_file()

语法高亮一个文件

132.12 highlight_string()

字符串的语法高亮

132.13 ignore_user_abort()

设置客户端断开连接时是否中断脚本的执行

132.14 pack()

把数据打包为二进制字符串

132.15 php_check_syntax()

检查 PHP 的语法（并执行）指定的文件

132.16 php_strip_whitespace()

返回删除注释和空格后的 PHP 源码

132.17 show_source()

别名 highlight_file

132.18 sleep()

延缓执行

132.19 sys_getloadavg()

获取系统的负载（load average）

132.20 time_nanosleep()

延缓执行若干秒和纳秒

132.21 time_sleep_until()

使脚本睡眠到指定的时间为止。

132.22 uniqid()

生成一个唯一 ID

132.23 unpack()

从二进制字符串中解包出数据。

132.24 usleep()

以指定的微秒数延迟执行

Part XXIX

Composer

Chapter 133

Overview

虽然 PHP 有很多库、框架和组件，但是 PHP 没有一个很好的方式来管理项目依赖，即使通过手动的方式去管理，依然需要解决自动加载器的问题。

Composer 是一个跨平台的 PHP 依赖管理工具，允许用户声明项目所依赖的代码库，并且可以在项目中安装它们。

Composer 不是一个包管理器。虽然涉及 “packages” 和 “libraries”，但是 composer 在每个项目的基础上进行管理，在指定项目的某个目录中（例如 vendor）进行安装。

具体来说，Composer 在本地安装一些资源包。一个包本质上就是一个包含东西的目录。通常情况下它存储 PHP 代码，但在理论上它可以是任何东西。并且它包含一个描述，其中有一个名称和一个版本号，这个名称和版本号用于识别该包。

默认情况下，Composer 不会在全局安装任何东西，因此它仅仅是一个依赖管理。

Composer 受到了 node npm 和 ruby's bundler 的强烈启发，可以这样为用户解决问题：

1. 你有一个项目依赖于若干个库。
2. 其中一些库依赖于其他库。
3. 声明所依赖的东西。

Composer 会找出哪个版本的包需要安装，并安装它们（将它们下载到你的项目中）。例如，正在创建的一个项目需要一个库来做日志记录，当前决定使用 monolog。为了将它添加到项目中，用户所需要做的就是创建一个 composer.json 文件，其中描述了项目的依赖关系。

composer.json 的内容指出当前的项目需要一些 monolog/monolog 的包，从 1.2 开始的任何版本。

```
{
   "require": {
      "monolog/monolog": "1.2.*"
   }
}
```

Composer 使用 PHP 开发，因此运行 Composer 需要 PHP 5.3.2+ 以上版本。一些敏感的 PHP 设置和编译标志也是必须的，但是对于任何不兼容项安装程序都会抛出警告。

目前支持 Composer 的框架包括 Symfony、Laravel、CodeIgniter、CakePHP、FuelPHP、Drupal 和 SilverStripe 等。

Security Advisories Checker 提供一个 web 服务和一个命令行工具，二者都会仔细检查你的 composer.lock 文件，并且告诉你任何你需要更新的依赖。

VersionEye web 服务可以监控你的 Github 及 BitBucket 帐号中的 composer.json 文件，并且当包有新更新时会发送邮件给你。

133.1 Installation

用户可以安装 Composer 到局部（当前工作目录）或是全局（例如/usr/local/bin）。

133.1.1 Project

在局部安装 Composer 时，首先要真正获取 Composer，同时这意味着需要把 Composer 下载到项目中。

```
$ curl -sS https://getcomposer.org/installer | php
```

如果上述方法由于某些原因失败了，还可以通过 php 下载安装器：

```
$ php -r "readfile('https://getcomposer.org/installer');" | php
```

接下来将检查一些 PHP 的设置，然后下载 composer.phar 到工作目录中。这是 Composer 的二进制文件，也是一个 PHAR 包（PHP 的归档），composer.phar 可以帮助用户在命令行中执行一些操作。

可以通过`--install-dir` 选项指定 Composer 的安装目录（它可以是一个绝对或相对路径）：

```
$ curl -sS https://getcomposer.org/installer | php -- --install-dir=bin
```

133.1.2 Global

如果把 composer.phar 保存到系统的 PATH 目录中就能在全局访问它，而且在类 Unix 系统中可以在使用时不加 php 前缀。

可以执行如下命令让 composer 在你的系统中进行全局调用：

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

现在只需要执行 `composer` 命令就可以使用Composer而不需要输入 `php composer.phar`。

Composer 是 homebrew-php 项目的一部分。

```
$ brew update
$ brew tap josegonzalez/homebrew-php
$ brew tap homebrew/versions
$ brew install php55-intl
$ brew install josegonzalez/php/composer
```

如果下载并且运行 Composer-Setup.exe，它将安装最新版本的 Composer 到 Windows 系统中，并设置好系统的环境变量，于是就可以在任何目录下直接使用 composer 命令。

如果需要手动安装 Composer，需要手动设置系统的环境变量 PATH 并运行安装命令下载 composer.phar 文件：

```
C:\Users\username>cd C:\bin
C:\bin>php -r "readfile('https://getcomposer.org/installer');" | php
```

如果收到 readfile 错误提示，需要使用 http 链接或者在 php.ini 中开启 php_openssl.dll 。

在 composer.phar 同级目录下新建文件 composer.bat ：

```
C:\bin>echo @php "%~dp0composer.phar" %*>composer.bat
```

关闭当前的命令行窗口，打开新的命令行窗口进行测试：

```
C:\Users\username>composer -V
Composer version 27d8904
```

要检查 Composer 是否正常工作，只需要通过 php 来执行 PHAR，正常情况下这将返回一个可执行的命令列表。

```
$ php composer.phar
```

Composer 的进程退出代码意义如下：

- 0: 正常
- 1: 通用/未知错误
- 2: 依赖关系处理错误

另外，也可以仅执行 `--check` 选项而无需下载 Composer。

```
$ curl -sS https://getcomposer.org/installer | php -- --help
```

PEAR 是另一个跟 Composer 很类似的依赖包管理器，但是也有一些显著的区别。

```
$ sudo apt-get install php-pear
$ pear install foo
```

首先，PEAR 需要扩展包有专属的结构，开发者在开发扩展包的时候要提前考虑为 PEAR 定制，否则后面将无法使用 PEAR。

其次，PEAR 安装扩展包的时候是全局安装的，安装了某个扩展包之后同一台服务器上的所有项目都能用上，好处是当多个项目共同使用同一个扩展包的同一个版本，坏处是如果需要使用不同版本就会产生冲突。

现在，Composer 可以用来安装 PEAR 扩展包，例如：

```
{
   "repositories": [
      {
          "type": "pear",
          "url": "http://pear2.php.net"
      }
   ],
   "require": {
      "pear-pear2/PEAR2_Text_Markdown": "*",
      "pear-pear2/PEAR2_HTTP_Request": "*"
   }
}
```

第一部分 "repositories" 是让 Composer 从哪个渠道去获取扩展包，然后"repositories" 部分使用如下的命名规范[1]：

```
pear-channel/Package
```

在成功安装扩展包以后，代码会放到项目的 vendor 文件夹中，并且可以通过加载 Composer 的自动加载器进行加载，例如：

```
vendor/pear-pear2.php.net/PEAR2_HTTP_Request/pear2/HTTP/Request.php
```

在代码里面可以这样使用：

```
<?php
$request = new pear2\HTTP\Request();
```

133.2 Dependence

Composer 通过一个 composer.json 文件持续地追踪项目依赖，用户可以手动管理 composer.json 或是使用 Composer 自己管理。

Composer 安装项目的依赖时，如果在当前目录下没有一个 composer.json 文件，需要手动创建，然后执行 `install` 命令来解决和下载依赖：

```
$ php composer.phar install
```

[1]前缀“pear”是为了避免冲突。

除此之外，`composer init` 命令将会引导用户创建一个完整的 composer.json 文件到项目中。

如果进行了全局安装，并且没有 phar 文件在当前目录，可以使用下面的命令代替：

```
$ composer install
```

上述命令将下载指定的依赖（例如 monolog）到 vendor 目录（例如 vendor/monolog/-monolog）目录。

Composer 也可以处理全局依赖和对应的二进制文件，只需要在命令前加上 global 前缀。例如，如果想安装 PHPUnit 并使它全局可用，可以运行下面的命令：

```
$ composer global require phpunit/phpunit
```

上述指令将创建一个 ~/.composer 目录存放全局依赖，而且添加 ~/.composer/vendor/bin 目录到 $PATH 变量让已安装依赖的二进制命令全局可用。

133.3 Repository

packagist 是 Composer 的主要资源库，在没有指定 package 的来源时默认来源就是从 packagist。

Packagist 是 Composer 主要的一个包信息存储库，它默认是启用的。任何在 packagist 上发布的包都可以直接被 Composer 使用。例如，monolog 被发布在 packagist 上，用户就可以直接使用它，而不必指定任何额外的来源信息。

一个 Composer 的库基本上是一个包的源：记录了可以得到包的地方。Packagist 的目标是成为大家使用库资源的中央存储平台。这意味着用户可以 require 那里的任何包。

当访问 packagist.org 时可以浏览和搜索资源包，而且可以通过向 packagist.org 提交自己的 package 的来源地址来让 packagist 抓取自己的包。一旦完成，自己的包将可以提供给任何人使用。

任何支持 Composer 的开源项目应该发布自己的包在 packagist 上，虽然并不一定要发布在 packagist 上来使用 Composer。

除了把资源包放在 packagist.org 上，也可以自行托管资源库。

- Private company packages
  如果你是一个公司的职员，对公司内部的资源包使用 composer，你可能会想让这些包保持私有的状态。
- Separate ecosystem
  如果你的项目有自己的生态系统，并且自己的资源包不需要被其它项目所复用，你可能会想将它们从 packagist.org 上分离出来。其中一个例子就是 wordpress 的插件。

对于自行托管的软件包，建议使用 composer 类型资源库设置，它将提供最佳的性能。

这里有一些工具，可以帮助你创建 composer 类型的资源库。

133.3.1 Packagist

packagist 的底层是开源的。这意味着你可以只安装你自己拷贝的 packagist，改造并使用它。这真的是很直接简单的事情。然而，由于其规模和复杂性，对于大多数中小型企业还是建议使用 Satis。

Packagist 是一个 Symfony2 应用程序，并且托管在 GitHub 上 github.com/composer/packagist。它内部使用了 composer 并作为 VCS 资源库与 composer 用户之间的代理。它拥有所有 VCS 资源包的列表，定期重新抓取它们，并将其作为一个 composer 资源库。

要设置你的副本，只需要按照 github.com/composer/packagist 的说明进行操作。

可以在 composer.json 中禁用默认的 Packagist 资源库。

```
{
    "repositories": [
        {
            "packagist": false
        }
    ]
}
```

133.3.2 Satis

Satis（https://github.com/composer/satis）是一个静态的 composer 资源库生成器。它像是一个超轻量级的、基于静态文件的 packagist 版本。

你给它一个包含 composer.json 的存储库，定义好 VCS 和资源库。它会获取所有你列出的包，并打印 packages.json 文件，作为 composer 类型的资源库。

133.3.3 Artifact

在某些情况下，或许没有能力拥有之前提到的任何一种线上资源库，而且大部分的时间它们都是私有的。

为了简化维护，可以简单的使用 artifact 资源库类型，来引用一个包含那些私有包的 ZIP 存档的文件夹：

```
{
    "repositories": [
        {
            "type": "artifact",
            "url": "path/to/directory/with/zips/"
        }
    ],
```

```
   "require": {
      "private-vendor-one/core": "15.6.2",
      "private-vendor-two/connectivity": "*",
      "acme-corp/parser": "10.3.5"
   }
}
```

每个 zip artifact 都只是一个 ZIP 存档，放置在 composer.json 所在的根目录：

```
$ unzip -l acme-corp-parser-10.3.5.zip

composer.json
...
```

如果有两个不同版本的资源包，它们都会被导入。当有一个新版本的存档被添加到 artifact 文件夹，并且你运行了 update 命令，该版本就会被导入，并且 Composer 将更新到最新版本。

133.4 Library

133.4.1 Package

每一个项目都是一个包（package），项目和库之间唯一的区别仅仅在于项目是一个没有名字的包。

只要有一个 composer.json 文件在目录中，那么整个目录就是一个包。

当添加一个 require 到项目中，就是在创建一个依赖于其它库的包。

为了使它成为一个可安装的包，需要给它一个名称。可以通过 composer.json 中的 name 来定义：

```
{
  "name": "acme/hello-world",
  "require": {
    "monolog/monolog": "1.0.*"
  }
}
```

在上述情况下，项目的名称为 acme/hello-world，其中 acme 是供应商的名称，供应商的名称是必须填写的。

如果不知道拿什么作为供应商的名称，那么使用 github 上的用户名通常是不错的选择。虽然包名不区分大小写，但惯例是使用小写字母，并用连字符作为单词的分隔。

事实上，在 composer 内部将每一个版本都视为一个单独的包。尽管在你使用 composer 时这种区别无关紧要，但当你想改变它时，这就显得至关重要。

除了名称和版本号，还存放了有用的元数据。与安装关系最密切的是 source 信息，它申明了在哪里可以获得资源包的内容。包数据指向包内容，并有两种指向方式：dist 和 source。

- Dist
  dist 指向一个存档，该存档是对一个资源包的某个版本的数据进行的打包。通常是已经发行的稳定版本。
- Source
  source 指向一个开发中的源。这通常是一个源代码仓库，例如 git。当你想要对下载下来的资源包进行修改时，可以这样获取。

你可以使用其中任意一个，或者同时使用。这取决于其它的一些因素，比如“user-supplied 选项”和“包的稳定性”，前者将会被优先考虑。

133.4.2 Repository

一个资源库是一个包的来源。它是一个 packages/versions 的列表。Composer 将查看所有你定义的 repositories 以找到你项目需要的资源包。

默认情况下已经将 Packagist.org 注册到 Composer。你可以在 composer.json 中申明更多的资源库，把它们加入你的项目中。

资源库的定义仅可用于“root 包”，而在你依赖的包中定义的资源库将不会被加载。

- composer
  主资源库的类型为 composer。它使用一个单一的 packages.json 文件，包含了所有的资源包元数据。
- vcs
  VCS 表示版本控制系统。这包括像 git、svn 或 hg 这样的版本管理系统。Composer 有一个资源类型可以从这些系统安装软件包。
- pear
  pear 类型资源库，使得从任何 PEAR 渠道安装资源包成为可能。Composer 将为所有此类型的包增加前缀（类似于 pear-{渠道名称}/）以避免冲突。而在之后使用别名时也增加前缀（如 pear-{渠道别名}/）。
- package
  如果需要使用一个项目，它无法通过上述任何一种方式支持 composer，你仍然可以使用 package 类型定义资源库。

composer 也是 packagist.org 所使用的资源类型。要引用一个 composer 资源库，只需要提供一个存放 packages.json 文件的目录路径。比如要引用 packagist.org 下的 /packages.json，它的 URL 就应该是 packagist.org。而 example.org/packages.json 的 URL 应该是 example.org。

唯一必须的字段是 packages。它的 JSON 结构如下：

```
{
```

```
   "packages": {
      "vendor/package-name": {
         "dev-master": { @composer.json },
         "1.0.x-dev": { @composer.json },
         "0.0.1": { @composer.json },
         "1.0.0": { @composer.json }
      }
   }
}
```

@composer.json 标记将会从此包的指定版本中读取 composer.json 的内容，其内至少应包含以下信息：

- name
- version
- dist or source

这是一个最简单的包定义：

```
{
   "name": "smarty/smarty",
   "version": "3.1.7",
   "dist": {
      "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
      "type": "zip"
   }
}
```

packages 的 JSON body 还可以包含任何在 composer.json 架构中介绍的字段。

notify-batch 字段允许你指定一个 URL，它将会在用户安装每一个包时被调用。该 URL 可以是（与其资源库相同域名的）绝对路径或者一个完整的 URL 地址。

```
{
   "notify-batch": "/downloads/"
}
```

对于 example.org/packages.json 包含的 monolog/monolog 包，它将会发送一个 POST 请求到 example.org/downloads/，使用下面的 JSON request body：

```
{
   "downloads": [
      {"name": "monolog/monolog", "version": "1.2.1.0"},
   ]
}
```

version 字段将包含标准化的版本号。

notify-batch 字段是可选的。

对于较大的资源库，可以拆分 packages.json 为多个文件，然后使用 includes 字段允许引用这些额外的文件。

```
{
   "includes": {
      "packages-2011.json": {
         "sha1": "525a85fb37edd1ad71040d429928c2c0edec9d17"
      },
      "packages-2012-01.json": {
         "sha1": "897cde726f8a3918faf27c803b336da223d400dd"
      },
      "packages-2012-02.json": {
         "sha1": "26f911ad717da26bbcac3f8f435280d13917efa5"
      }
   }
}
```

文件的 SHA-1 码允许它被缓存，仅在 hash 值改变时重新请求。

includes 字段是可选的。你也许并不需要它来自定义存储库。

对于非常大的资源库，像 packagist.org 使用 so-called provider 文件是首选方法。

provider-includes 字段允许你设置一个列表，来申明这个资源库提供的包名称。在这种情况下文件的哈希算法必须使用 sha256。

providers-url 描述了如何在服务器上找到这些 provider 文件。它是以资源库的根目录为起点的绝对路径。

```
{
   "provider-includes": {
      "providers-a.json": {
         "sha256": "f5b4bc0b354108ef08614e569c1ed01a2782e67641744864a74e788982886f4c"
      },
      "providers-b.json": {
         "sha256": "b38372163fac0573053536f5b8ef11b86f804ea8b016d239e706191203f6efac"
      }
   },
   "providers-url": "/p/%package%$%hash%.json"
}
```

这些文件包含资源包的名称以及哈希值，以验证文件的完整性，例如：

```
{
   "providers": {
      "acme/foo": {
```

```
            "sha256": "38968de1305c2e17f4de33aea164515bc787c42c7e2d6e25948539a14268bb82"
        },
        "acme/bar": {
            "sha256": "4dd24c930bd6e1103251306d6336ac813b563a220d9ca14f4743c032fb047233"
        }
    }
}
```

上述文件申明了 acme/foo 和 acme/bar 可以在这个资源库找到，通过加载由 providers-url 引用的文件，替换%package% 为包名并且替换%hash% 为 sha256 的值。这些文件本身只包含上文提到的 packages 的定义。

这些字段是可选的。你也许并不需要它们来自定义存储库。

packages.json 文件是用一个 PHP 流加载的。你可以使用 options 参数来设定额外的流信息。你可以设置任何有效的 PHP 流上下文选项。

从 VCS 资源库加载一个包时，最常见的是维护自己 fork 的第三方库。如果你在项目中使用某些库，并且你决定改变这些库内的某些东西，你会希望你项目中使用的是你自己的修正版本。如果这个库是在 GitHub 上（这种情况经常出现），你可以简单的 fork 它并 push 你的变更到这个 fork 里。在这之后你更新项目的 composer.json 文件，添加你的 fork 作为一个资源库，变更版本约束来指向你的自定义分支。

例如，假设你 fork 了 monolog，在 bugfix 分支修复了一个 bug：

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "https://github.com/igorw/monolog"
        }
    ],
    "require": {
        "monolog/monolog": "dev-bugfix"
    }
}
```

当你运行 php composer.phar update 时，你应该得到你修改的版本，而不是 packagist.org 上的 monolog/monolog。

注意，你不应该对包进行重命名，除非你真的打算摆脱原来的包，并长期的使用你自己的 fork。这样 Composer 就会正确获取你的包了。如果你确定要重命名这个包，你应该在默认分支（通常是 master 分支）上操作，而不是特性分支，因为包的名字取自默认分支。

如果其它包依赖你 fork 的这个分支，可能要对它做版本号的行内别名设置，才能够准确的识别版本约束。

使用私有资源库和使用第三方资源库是完全相同的解决方案，也可以让你使用你GitHub和 BitBucket 上的私人代码库进行工作：

```
{
    "require": {
        "vendor/my-private-repo": "dev-master"
    },
    "repositories": [
        {
            "type": "vcs",
            "url": "git@bitbucket.org:vendor/my-private-repo.git"
        }
    ]
}
```

唯一的要求是为一个 git 客户端安装 SSH 秘钥。

Git 并不是 VCS 资源库唯一支持的版本管理系统，还包括 Subversion 和 Mercurial 等。

为了从这些系统获取资源包，你必须安装对应的客户端，这可能是不方便的。基于这个原因，这里提供了 GitHub 和 BitBucket 的 API 的特殊支持，以便在无需安装版本控制系统的情况下获取资源包。在 VCS 资源库提供的 dist 中获取 zip 存档。

VCS 驱动将基于 URL 自动检测版本库类型。但如果可能，你需要明确的指定一个 git、svn 或 hg 作为资源库类型，而不是 vcs。

If you set the no-api key to true on a github repository it will clone the repository as it would with any other git repository instead of using the GitHub API. But unlike using the git driver directly, composer will still attempt to use github's zip files.

由于 Subversion 没有原生的分支和标签的概念，Composer 假设在默认情况下该代码位于 $url/trunk、$url/branches 和 $url/tags 内。如果你的存储库使用了不同的布局，你可以更改这些值。例如，如果你使用大写的名称，你可以像这样配置资源库：

```
{
    "repositories": [
        {
            "type": "vcs",
            "url": "http://svn.example.org/projectA/",
            "trunk-path": "Trunk",
            "branches-path": "Branches",
            "tags-path": "Tags"
        }
    ]
}
```

如果你的存储库目录中没有任何分支或标签文件夹，你可以将 branches-path 或 tags-path 设置为 false。

如果是一个位于子目录的包，例如，/trunk/foo/bar/composer.json 和 /tags/1.0/foo/bar/composer.json，那么你可以让 composer 通过”package-path” 选项设置的子目录进行访问，在这个例子中可以将其设置为”package-path”: ”foo/bar/”。

pear 类型资源库，使得从任何 PEAR 渠道安装资源包成为可能。Composer 将为所有此类型的包增加前缀（类似于 pear-{渠道名称}/）以避免冲突。而在之后使用别名时也增加前缀（如 pear-{渠道别名}/）。

例如，使用 pear2.php.net：

```
{
   "repositories": [
      {
         "type": "pear",
         "url": "http://pear2.php.net"
      }
   ],
   "require": {
      "pear-pear2.php.net/PEAR2_Text_Markdown": "*",
      "pear-pear2/PEAR2_HTTP_Request": "*"
   }
}
```

在这种情况下渠道的简称（别名）是 pear2，因此 PEAR2_HTTP_Request 包的名称应该写作 pear-pear2/PEAR2_HTTP_Request。

pear 类型的资源库对每个 requires 都要做完整的请求，因此可能大大降低安装速度。

通过自定义供应商名称，对 PEAR 渠道包进行别名是允许的。例如，假设你有一个私人 PEAR 库，并希望使用 Composer 从 VCS 集成依赖。你的 PEAR 库包含以下资源包：

- BasePackage。
- IntermediatePackage 依赖于 BasePackage。
- TopLevelPackage1 和 TopLevelPackage2 都依赖于 IntermediatePackage。

如果没有一个供应商别名，Composer 将使用 PEAR 渠道名称作为包名的一部分：

- pear-pear.foobar.repo/BasePackage
- pear-pear.foobar.repo/IntermediatePackage
- pear-pear.foobar.repo/TopLevelPackage1
- pear-pear.foobar.repo/TopLevelPackage2

假设之后的某个时间，你希望将你的 PEAR 包迁移，使用 Composer 资源库和命名方案，并且采用 foobar 作为供应商名称。这样之前使用 PEAR 包的项目将不会看到更新的资源包，因为它们有不同的供应商名称(foobar/IntermediatePackage 与 pear-pear.foobar.repo/IntermediatePackage)。

你可以通过从一开始就为 PEAR 资源库指定 vendor-alias 来避免这种情况的发生，以得到一个不会过时的包名。

为了说明这一点，下面的例子会从你的 PEAR 资源库中得到 BasePackage、TopLevelPackage1 和 TopLevelPackage2 资源包，并从 Github 资源库中获取 IntermediatePackage 资源包：

```
{
   "repositories": [
      {
         "type": "git",
         "url": "https://github.com/foobar/intermediate.git"
      },
      {
         "type": "pear",
         "url": "http://pear.foobar.repo",
         "vendor-alias": "foobar"
      }
   ],
   "require": {
      "foobar/TopLevelPackage1": "*",
      "foobar/TopLevelPackage2": "*"
   }
}
```

如果你想使用一个项目，它无法通过上述任何一种方式支持 composer，你仍然可以使用 package 类型定义资源库。

基本上，你可以定义与 packages.json 中 composer 类型资源库相同的信息，但需要为每个这样的资源包分别定义。同样，至少应该包含以下信息：name、version、(dist 或 source)。

这是一个 smarty 模板引擎的例子：

```
{
   "repositories": [
      {
         "type": "package",
         "package": {
            "name": "smarty/smarty",
            "version": "3.1.7",
            "dist": {
               "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
               "type": "zip"
            },
            "source": {
               "url": "http://smarty-php.googlecode.com/svn/",
```

```
                "type": "svn",
                "reference": "tags/Smarty_3_1_7/distribution/"
            },
            "autoload": {
                "classmap": ["libs/"]
            }
        }
    }
  ],
  "require": {
     "smarty/smarty": "3.1.*"
  }
}
```

通常你不需要去定义 source，因为你并不是真的需要它。

注意，package 资源库类型存在以下限制，因此应尽可能避免使用：

- Composer 将不会更新资源包，除非你修改了 version 字段。
- Composer 将不会更新 commit references，因此如果你使用 master reference，将不得不删除该程序包以强制更新，并且将不得不面对一个不稳定的 lock 文件。

133.4.3 Platform

Composer 将那些已经安装在系统上，但并不是由 Composer 安装的包视为一个虚拟的平台软件包（包括 PHP 本身，PHP 扩展和一些系统库等）。

- php
  php 表示用户的 PHP 版本要求，可以对其做出限制（例如 >=5.4.0）。如果需要 64 位版本的 PHP，可以使用 php-64bit 进行限制。
- hhvm
  hhvm 代表的是 HHVM（也就是 HipHop Virtual Machine）运行环境的版本，并且允许设置一个版本限制，例如`'>=2.3.3'`。
- ext-<name>
  ext-<name> 可以帮助指定需要的 PHP 扩展（包括核心扩展）。通常 PHP 拓展的版本可以是不一致的，将它们的版本约束为 * 是一个不错的主意。一个 PHP 扩展包的例子就是包名可以写成 ext-gd。
- lib-<name>
  lib-<name> 允许对 PHP 库的版本进行限制，可供使用的名称包括 curl、iconv、icu、libxml、openssl、pcre、uuid、xsl。

可以使用 `composer show --platform` 命令来获取可用的平台软件包的列表。

```
$ composer show --platform
```

```
composer-plugin-api 1.1.0 The Composer Plugin API
ext-calendar     7.0.13   The calendar PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-ctype        7.0.13   The ctype PHP extension (actual version: 7.0.13-1+deb.sury.org
    ~trusty+1)
ext-curl         7.0.13   The curl PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-date         7.0.13   The date PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-dom          20031129 The dom PHP extension
ext-exif         0        The exif PHP extension (actual version: 1.4 $Id: 8
    bdc0c8f27c2c9dd1f7551f1f9fe3ab57a06a4b1 $)
ext-fileinfo     1.0.5    The fileinfo PHP extension
ext-filter       7.0.13   The filter PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-ftp          7.0.13   The ftp PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-gd           7.0.13   The gd PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-gettext      7.0.13   The gettext PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-hash         1.0      The hash PHP extension
ext-iconv        7.0.13   The iconv PHP extension (actual version: 7.0.13-1+deb.sury.org
    ~trusty+1)
ext-igbinary     1.2.2-dev The igbinary PHP extension
ext-json         1.4.0    The json PHP extension
ext-libxml       7.0.13   The libxml PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-mbstring     7.0.13   The mbstring PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-memcache     3.0.9-dev The memcache PHP extension
ext-memcached    3.0.0b1  The memcached PHP extension
ext-msgpack      2.0.1    The msgpack PHP extension
ext-mysqli       7.0.13   The mysqli PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-mysqlnd      0        The mysqlnd PHP extension (actual version: mysqlnd 5.0.12-dev
    - 20150407 - $Id: 241ae00989d1995ffcbbf63d579943635faf9972 $)
ext-openssl      7.0.13   The openssl PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-pcntl        7.0.13   The pcntl PHP extension (actual version: 7.0.13-1+deb.sury.org
    ~trusty+1)
```

```
ext-pcre         7.0.13  The pcre PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-PDO          7.0.13  The PDO PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-pdo_mysql    7.0.13  The pdo_mysql PHP extension (actual version: 7.0.13-1+deb.sury
    .org~trusty+1)
ext-phalcon      3.0.2   The phalcon PHP extension
ext-Phar         2.0.2   The Phar PHP extension
ext-posix        7.0.13  The posix PHP extension (actual version: 7.0.13-1+deb.sury.org
    ~trusty+1)
ext-readline     7.0.13  The readline PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-redis        2.2.8   The redis PHP extension (actual version: 2.2.8-devphp7)
ext-Reflection   7.0.13  The Reflection PHP extension (actual version: 7.0.13-1+deb.
    sury.org~trusty+1)
ext-session      7.0.13  The session PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-shmop        7.0.13  The shmop PHP extension (actual version: 7.0.13-1+deb.sury.org
    ~trusty+1)
ext-SimpleXML    7.0.13  The SimpleXML PHP extension (actual version: 7.0.13-1+deb.sury
    .org~trusty+1)
ext-sockets      7.0.13  The sockets PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-SPL          7.0.13  The SPL PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-sysvmsg      7.0.13  The sysvmsg PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-sysvsem      7.0.13  The sysvsem PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-sysvshm      7.0.13  The sysvshm PHP extension (actual version: 7.0.13-1+deb.sury.
    org~trusty+1)
ext-tokenizer    7.0.13  The tokenizer PHP extension (actual version: 7.0.13-1+deb.sury
    .org~trusty+1)
ext-wddx         7.0.13  The wddx PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-xml          7.0.13  The xml PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-xmlreader    7.0.13  The xmlreader PHP extension (actual version: 7.0.13-1+deb.sury
    .org~trusty+1)
ext-xmlwriter    7.0.13  The xmlwriter PHP extension (actual version: 7.0.13-1+deb.sury
    .org~trusty+1)
```

```
ext-xsl          7.0.13   The xsl PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
ext-yaf          3.0.4    The yaf PHP extension
ext-Zend-OPcache 7.0.13   The Zend OPcache PHP extension (actual version: 7.0.13-1+deb.
    sury.org~trusty+1)
ext-zip          1.13.5   The zip PHP extension
ext-zlib         7.0.13   The zlib PHP extension (actual version: 7.0.13-1+deb.sury.org~
    trusty+1)
lib-curl         7.35.0   The curl PHP library
lib-iconv        2.19     The iconv PHP library
lib-libxml       2.9.1    The libxml PHP library
lib-openssl      1.0.2.10 OpenSSL 1.0.2j 26 Sep 2016
lib-pcre         8.39     The pcre PHP library
lib-xsl          1.1.28   The xsl PHP library
php              7.0.13   The PHP interpreter
php-64bit        7.0.13   The PHP interpreter, 64bit
php-ipv6         7.0.13   The PHP interpreter with IPv6 support
```

133.4.4 Version

在开发自己的 Package 时，需要一些方法来指明自己开发的包的版本。

在 Packagist 上发布自己的包时，Packagist 能够从 VCS (git, svn, hg) 的信息推断出包的版本，因此不必自己手动指明版本号，并且也不建议这样做。

如果想要手动创建并且真的要明确指定它，只需要添加一个 version 字段：

```
{
   "version": "1.0.0"
}
```

实际上，应该尽量避免手动设置版本号，因为标签的值必须与标签名相匹配。

133.4.5 Tagging

对于每一个看起来像版本号的标签，都会相应的创建一个包的版本。它应该符合`'X.Y.Z'`或者`'vX.Y.Z'`的形式，`-patch`、`-alpha`、`-beta`或`-RC`这些后缀是可选的。在后缀之后也可以再跟上一个数字。

- 1.0.0
- v1.0.0
- 1.10.5-RC1
- v4.4.4beta2
- v2.0.0-alpha

- v2.0.4-p1

即使标签带有前缀 v，由于在需要 require 一个版本的约束时是不允许这种前缀的，因此 v 将被省略（例如标签 V1.0.0 将创建 1.0.0 版本）。

133.4.6 Branch

对于每一个分支，都会相应的创建一个包的开发版本。如果分支名看起来像一个版本号，那么将创建一个如同 {分支名}-dev 的包版本号。例如，一个分支 2.0 将产生一个 2.0.x-dev 包版本（加入了.x 是出于技术的原因，以确保它被识别为一个分支，而 2.0.x 的分支名称也是允许的，它同样会被转换为 2.0.x-dev）。

如果分支名看起来不像一个版本号，它将会创建 dev-{分支名} 形式的版本号。例如，master 将产生一个 dev-master 的版本号。

- 1.x
- 1.0 (equals 1.0.x)
- 1.1.x

当安装一个新的版本时，将会自动从它的 source 中拉取。

133.4.7 Aliases

别名表示一个包版本的别名。例如，可以为 dev-master 设置别名 1.0.x-dev，这样就可以通过 `require 1.0.x-dev` 来得到 dev-master 版本的包。

Chapter 134

Configuration

134.1 composer.json

要开始在项目中使用 Composer，只需要一个 composer.json 文件，该文件包含了项目的依赖和其它的一些元数据，而且允许定义嵌套结构。

只要有一个包含 composer.json 文件的库存储在线上版本控制系统 (例如 Git) 中，用户自己的库就可以被 Composer 所安装。例如，可以将 acme/hello-world 库发布到 github.com/username/hello-world 中。

现在测试这个 acme/hello-world 包，我们在本地创建一个新的项目并命名为 acme/blog，这个博客依赖 acme/hello-world，而后者又依赖 monolog/monolog。我们可以在某处创建一个新的 blog 文件夹来完成它，并且需要包含 composer.json 文件：

```
{
    "name": "acme/blog",
    "require": {
        "acme/hello-world": "dev-master"
    }
}
```

在这个例子中 name 不是必须的，因为我们并不想将它发布为一个库。在这里为 composer.json 文件添加描述。

现在我们需要告诉我们的应用，在哪里可以找到 hello-world 的依赖，为此我们需要在 composer.json 中添加 repositories 来源申明：

```
{
    "name": "acme/blog",
    "repositories": [
        {
```

```
            "type": "vcs",
            "url": "https://github.com/username/hello-world"
        }
    ],
    "require": {
        "acme/hello-world": "dev-master"
    }
}
```

接下来，现在就可以使用 Composer 的 install 命令来安装你的依赖包。

实际上，任何含有 composer.json 的 GIT、SVN、HG 存储库，都可以通过 require 字段指定“包来源”和“声明依赖”来添加到实际应用项目中。

Composer 有一个 JSON schema 格式化文档被用来验证用户的 composer.json 文件。事实上，它已经被 validate 命令所使用。

134.1.1 root

“root 包”是指由 composer.json 定义的在你项目根目录的包。这是 composer.json 定义你项目所需的主要条件。(简单的说，你自己的项目就是一个 root 包)

某些字段仅适用于“root 包”上下文。config 字段就是其中一个例子。只有“root 包”可以定义。在依赖包中定义的 config 字段将被忽略，这使得 config 字段只有“root 包”可用(root-only)。

如果你克隆了其中的一个依赖包，直接在其上开始工作，那么它就变成了“root 包”。与作为他人的依赖包时使用相同的 composer.json 文件，但上下文发生了变化。

一个资源包是不是“root 包”，取决于它的上下文。例如，如果你的项目依赖 monolog 库，那么你的项目就是“root 包”。但是，如果你从 GitHub 上克隆了 monolog 为它修复 bug，那么此时 monolog 就是“root 包”。

134.1.2 name

name 是包的名称，它包括供应商名称和项目名称，使用/分隔。例如，monolog/monolog 或 igorw/event-source。

通常容易产生相同的项目名称，而供应商名称的存在则很好的解决了命名冲突的问题。例如，它允许两个不同的人创建同样名为 json 的库，而之后它们将被命名为 igorw/json 和 seldaek/json。

这里，我们需要引入 monolog/monolog，供应商名称与项目的名称相同，对于一个具有唯一名称的项目，我们推荐这么做。它还允许以后在同一个命名空间添加更多的相关项目。如果你维护着一个库，这将使你可以很容易的把它分离成更小的部分。

对于需要发布的包（库），这是必须填写的。

134.1.3 description

一个包的简短描述。通常这个最长只有一行。

对于需要发布的包（库），这是必须填写的。

134.1.4 version

version 不是必须的，并且建议忽略（见下文）。

它应该符合 `'X.Y.Z'` 或者 `'vX.Y.Z'` 的形式，`-dev`、`-patch`、`-alpha`、`-beta` 或 `-RC` 这些后缀是可选的。在后缀之后也可以再跟上一个数字。

在前面的例子中引入的 monolog 版本指定为 1.0.*。这表示任何从 1.0 开始的开发分支，它将会匹配 1.0.0、1.0.2 或者 1.0.20。

版本约束可以用几个不同的方法来指定。

通常，我们能够从 VCS (git, svn, hg) 的信息推断出包的版本号，在这种情况下，我们建议忽略 version。

Packagist 使用 VCS 仓库，因此 version 定义的版本号必须是真实准确的。自己手动指定的 version，最终有可能在某个时候因为人为错误造成问题。

134.1.5 type

包的安装类型，默认为 library。

包的安装类型，用来定义安装逻辑。如果你有一个包需要一个特殊的逻辑，你可以设定一个自定义的类型。这可以是一个 symfony-bundle，一个 wordpress-plugin 或者一个 typo3-module。这些类型都将是具体到某一个项目，而对应的项目将要提供一种能够安装该类型包的安装程序。

composer 原生支持以下 4 种类型：

- library: 这是默认类型，它会简单的将文件复制到 vendor 目录。
- project: 这表示当前包是一个项目，而不是一个库。例如：框架应用程序 Symfony standard edition，内容管理系统 SilverStripe installer 或者完全成熟的分布式应用程序。使用 IDE 创建一个新的工作区时，这可以为其提供项目列表的初始化。
- metapackage: 当一个空的包，包含依赖并且需要触发依赖的安装，这将不会对系统写入额外的文件。因此这种安装类型并不需要一个 dist 或 source。
- composer-plugin: 一个安装类型为 composer-plugin 的包，它有一个自定义安装类型，可以为其它包提供一个 installler。

仅在你需要一个自定义的安装逻辑时才使用它。建议忽略这个属性，采用默认的 library。

134.1.6 keywords

该包相关的关键词的数组(可选), 这些可用于搜索和过滤, 例如 logging、events、database、redis 或 templating 等。

134.1.7 homepage

该项目网站的 URL 地址（可选)。

134.1.8 time

版本发布时间（可选)，必须符合 YYYY-MM-DD 或 YYYY-MM-DD HH:MM:SS 格式。

134.1.9 license

包的许可协议（可选)，它可以是一个字符串或者字符串数组，强烈建议提供此内容。

最常见的许可协议的推荐写法（按字母排序):

- Apache-2.0
- BSD-2-Clause
- BSD-3-Clause
- BSD-4-Clause
- GPL-2.0
- GPL-2.0+
- GPL-3.0
- GPL-3.0+
- LGPL-2.1
- LGPL-2.1+
- LGPL-3.0
- LGPL-3.0+
- MIT

对于闭源软件，你必须使用”proprietary” 协议标识符。

对于一个包，当允许在多个许可协议间进行选择时（”disjunctive license”)，这些协议标识符可以被指定为数组。

```
{
   "license": [
     "LGPL-2.1",
     "GPL-3.0+"
   ]
}
```

另外它们也可以由”or” 分隔，并写在括号中：

```
{
   "license": "(LGPL-2.1 or GPL-3.0+)"
}
```

同样，当有多个许可协议需要结合使用时（”conjunctive license”），它们应该被”and” 分隔，并写在括号中。

134.1.10 authors

包的作者（可选）。这是一个对象数组，强烈建议提供此内容。

这个对象必须包含以下属性：

- name: 作者的姓名，通常使用真名。
- email: 作者的 email 地址。
- homepage: 作者主页的 URL 地址。
- role: 该作者在此项目中担任的角色（例如开发人员或翻译）。

```
{
   "authors": [
      {
         "name": "Nils Adermann",
         "email": "naderman@naderman.de",
         "homepage": "http://www.naderman.de",
         "role": "Developer"
      },
      {
         "name": "Jordi Boggiano",
         "email": "j.boggiano@seld.be",
         "homepage": "http://seld.be",
         "role": "Developer"
      }
   ]
}
```

134.1.11 support

获取项目支持的向相关信息对象（可选）。

这个对象必须包含以下属性：

- email: 项目支持 email 地址。
- issues: 跟踪问题的 URL 地址。

- forum: 论坛地址。
- wiki: Wiki 地址。
- irc: IRC 聊天频道地址，类似于 irc://server/channel。
- source: 网址浏览或下载源。

```
{
   "support": {
      "email": "support@example.org",
      "irc": "irc://irc.freenode.org/composer"
   }
}
```

134.1.12 require

第一件事情 (并且往往只需要做这一件事) 就是需要在 composer.json 文件中指定 require 的值，也就是简单的告诉 Composer 项目中需要依赖哪些包。

```
{
  "require": {
     "monolog/monolog": "1.0.*"
  }
}
```

require 表示必须的软件包列表，除非这些依赖被满足，否则不会完成安装。

require 需要一个包名称（例如 monolog/monolog）映射到包版本（例如 1.0.*）的对象。

require 和 require-dev 还支持稳定性标签（@，仅针对“root 包”）。这允许你在 minimum-stability 设定的范围外做进一步的限制或扩展。例：如果你想允许依赖一个不稳定的包，你可以在一个包的版本约束后使用它，或者是一个空的版本约束内使用它。

```
{
   "require": {
      "monolog/monolog": "1.0.*@beta",
      "acme/foo": "@dev"
   }
}
```

如果你的依赖之一，有对另一个不稳定包的依赖，你最好在 require 中显示的定义它，并带上足够详细的稳定性标识。

```
{
   "require": {
      "doctrine/doctrine-fixtures-bundle": "dev-master",
      "doctrine/data-fixtures": "@dev"
```

```
    }
}
```

require 和 require-dev 还支持对 dev（开发）版本的明确引用（即版本控制系统中的提交编号 commit），以确保它们被锁定到一个给定的状态，即使你运行了更新命令。你只需要明确一个开发版本号，并带上诸如 #<ref> 的标识。

```
{
    "require": {
        "monolog/monolog": "dev-master#2eb0c0978d290a1c45346a1955188929cb4e5db7",
        "acme/foo": "1.0.x-dev#abc123"
    }
}
```

虽然这有时很方便，但不应该长期在你的包中使用，因为它有一个技术上的限制。composer.json 将仍然在哈希值之前指定的分支名称读取元数据，正因为如此，在某些情况下，它不会是一个实用的解决方法，如果可能，你应该总是尝试切换到拥有标签的版本。

require 也可以应用于行内别名，这样它将匹配一个约束，否则不会。

134.1.13 require-dev

require-dev 列表是为开发或测试等目的，额外列出的依赖。“root 包”的 require-dev 默认是会被安装的。然而 install 或 update 支持使用`--no-dev`参数来跳过 require-dev 字段中列出的包。

134.1.14 conflict

此列表中的包与当前包的这个版本冲突。它们将不允许同时被安装。

注意，在 conflict 中指定类似于 <1.0, >= 1.1 的版本范围时，这表示它与小于 1.0 并且同时大等于 1.1 的版本冲突，这很可能不是你想要的。在这种情况下你可能想要表达的是 <1.0 | >= 1.1 。

134.1.15 replace

这个列表中的包将被当前包取代。这使你可以 fork 一个包，以不同的名称和版本号发布，同时要求依赖于原包的其它包，在这之后依赖于你 fork 的这个包，因为它取代了原来的包。

这对于创建一个内部包含子包的主包也非常的有用。例如 symfony/symfony 这个主包，包含了所有 Symfony 的组件，而这些组件又可以作为单独的包进行发布。如果你 require 了主包，那么它就会自动完成其下各个组件的任务，因为主包取代了子包。

注意，在使用上述方法取代子包时，通常你应该只对子包使用 self.version 这一个版本约束，以确保主包仅替换掉子包的准确版本，而不是任何其他版本。

134.1.16 provide

List of other packages that are provided by this package. This is mostly useful for common interfaces. A package could depend on some virtual logger package, any library that implements this logger interface would simply list it in provide.

134.1.17 suggest

建议安装的包，它们增强或能够与当前包良好的工作。这些只是信息，并显示在依赖包安装完成之后，给你的用户一个建议，他们可以添加更多的包。

格式如下，版本约束变成了描述信息。

```
{
   "suggest": {
      "monolog/monolog": "Allows more advanced logging of the application flow"
   }
}
```

134.1.18 autoload

PHP autoloader 的自动加载映射。

Currently PSR-0 autoloading, PSR-4 autoloading, classmap generation and files includes are supported. PSR-4 is the recommended way though since it offers greater ease of use (no need to regenerate the autoloader when you add classes).

134.1.19 psr-4

Under the psr-4 key you define a mapping from namespaces to paths, relative to the package root. When autoloading a class like Foo\\Bar\\Baz a namespace prefix Foo\\pointing to a directory src/ means that the autoloader will look for a file named src/Bar/Baz.php and include it if present. Note that as opposed to the older PSR-0 style, the prefix (Foo\\) is not present in the file path.

Namespace prefixes must end in \\to avoid conflicts between similar prefixes. For example Foo would match classes in the FooBar namespace so the trailing backslashes solve the problem: Foo\\and FooBar\\are distinct.

The PSR-4 references are all combined, during install/update, into a single key => value array which may be found in the generated file vendor/composer/autoload_psr4.php.

```
{
    "autoload": {
        "psr-4": {
            "Monolog\\": "src/",
            "Vendor\\Namespace\\": ""
        }
    }
}
```

If you need to search for a same prefix in multiple directories, you can specify them as an array as such:

```
{
    "autoload": {
        "psr-4": { "Monolog\\": ["src/", "lib/"] }
    }
}
```

If you want to have a fallback directory where any namespace will be looked for, you can use an empty prefix like:

```
{
    "autoload": {
        "psr-4": { "": "src/" }
    }
}
```

134.1.20 psr-0

在 psr-0 下你定义了一个命名空间到实际路径的映射（相对于包的根目录）。注意，这里同样支持 PEAR-style 方式的约定（与命名空间不同，PEAR 类库在类名上采用了下划线分隔）。

请注意，命名空间的申明应该以\\结束，以确保 autoloader 能够准确响应。例如，Foo 将会与 FooBar 匹配，然而以反斜杠结束就可以解决这样的问题，Foo\\和 FooBar\\将会被区分开来。

在 install/update 过程中，PSR-0 引用都将被结合为一个单一的键值对数组，存储至 vendor/composer/autoload_namespaces.php 文件中。

```
{
    "autoload": {
        "psr-0": {
            "Monolog\\": "src/",
```

```
            "Vendor\\Namespace\\": "src/",
            "Vendor_Namespace_": "src/"
        }
    }
}
```

如果你需要搜索多个目录中一个相同的前缀，你可以将它们指定为一个数组。

```
{
    "autoload": {
        "psr-0": { "Monolog\\": ["src/", "lib/"] }
    }
}
```

PSR-0 方式并不仅限于申明命名空间，也可以是精确到类级别的指定。这对于只有一个类在全局命名空间的类库是非常有用的（如果 php 源文件也位于包的根目录)。例如，可以这样申明：

```
{
    "autoload": {
        "psr-0": { "UniqueGlobalClass": "" }
    }
}
```

如果你想设置一个目录作为任何命名空间的备用目录，你可以使用空的前缀，例如：

```
{
    "autoload": {
        "psr-0": { "": "src/" }
    }
}
```

134.1.21 classmap

classmap 引用的所有组合，都会在 install/update 过程中生成，并存储到 vendor/composer/autoload_classmap.php 文件中。这个 map 是经过扫描指定目录（同样支持直接精确到文件）中所有的.php 和.inc 文件里内置的类而得到的。

你可以用 classmap 生成支持支持自定义加载的不遵循 PSR-0/4 规范的类库。要配置它指向需要的目录，以便能够准确搜索到类文件。

```
{
    "autoload": {
        "classmap": ["src/", "lib/", "Something.php"]
    }
```

```
}
```

134.1.22 files

如果你想要明确的指定，在每次请求时都要载入某些文件，那么你可以使用'files' autoloading。通常作为函数库的载入方式（而非类库)。

```
{
   "autoload": {
      "files": ["src/MyLibrary/functions.php"]
   }
}
```

134.1.23 autoload-dev

This section allows to define autoload rules for development purposes.

Classes needed to run the test suite should not be included in the main autoload rules to avoid polluting the autoloader in production and when other people use your package as a dependency.

Therefore, it is a good idea to rely on a dedicated path for your unit tests and to add it within the autoload-dev section.

```
{
   "autoload": {
      "psr-4": { "MyLibrary\\": "src/" }
   },
   "autoload-dev": {
      "psr-4": { "MyLibrary\\Tests\\": "tests/" }
   }
}
```

134.1.24 include-path

这是目前唯一支持传统项目的做法，所有新的代码都建议使用自动加载。这是一个过时的做法，但 Composer 将仍然保留这个功能。

include-path 表示一个追加到 PHP include_path 中的列表（可选)。

```
{
   "include-path": ["lib/"]
}
```

134.1.25 target-dir

This is only present to support legacy PSR-0 style autoloading, and all new code should preferably use PSR-4 without target-dir and projects using PSR-0 with PHP namespaces are encouraged to migrate to PSR-4 instead.

定义当前包安装的目标文件夹（可选）。

若某个包的根目录，在它申明的命名空间之下，将不能正确的使用自动加载。而 target-dir 解决了这个问题。

Symfony 就是一个例子。它有一些独立的包作为组件。Yaml 组件就放在 Symfony\Component\Yaml 目录下，然而这个包的根目录实际上是 Yaml。为了使自动加载成为可能，我们需要确保它不会被安装到 vendor/symfony/yaml，而是安装到 vendor/symfony/yaml/Symfony/Component/Yaml，从而使 Symfony 定义的 autoloader 可以从 vendor/symfony/yaml 加载它。

要做到这一点 autoload 和 target-dir 应该定义如下：

```
{
   "autoload": {
      "psr-0": { "Symfony\\Component\\Yaml\\": "" }
   },
   "target-dir": "Symfony/Component/Yaml"
}
```

134.1.26 minimum-stability

这定义了通过稳定性过滤包的默认行为。默认为 stable（稳定）。因此如果你依赖于一个 dev（开发）包，你应该明确的进行定义。

对每个包的所有版本都会进行稳定性检查，而低于 minimum-stability 所设定的最低稳定性的版本，将在解决依赖关系时被忽略。对于个别包的特殊稳定性要求，可以在 require 或 require-dev 中设定。

可用的稳定性标识（按字母排序）：dev、alpha、beta、RC、stable。

134.1.27 prefer-stable

当此选项被激活时，Composer 将优先使用更稳定的包版本，可以使用`"prefer-stable": true` 来激活它。

134.1.28 repositories

使用自定义的包资源库。

默认情况下 composer 只使用 packagist 作为包的资源库。通过指定资源库，你可以从其他地方获取资源包。

Repositories 并不是递归调用的，只能在“Root 包”的 composer.json 中定义。附属包中的 composer.json 将被忽略。

支持以下类型的包资源库：

- composer: 一个 composer 类型的资源库，是一个简单的网络服务器（HTTP、FTP、SSH）上的 packages.json 文件，它包含一个 composer.json 对象的列表，有额外的 dist 和/或 source 信息。

  这个 packages.json 文件是用一个 PHP 流加载的。你可以使用 options 参数来设定额外的流信息。
- vcs: 从 git、svn 和 hg 取得资源。
- pear: 从 pear 获取资源。
- package: 如果你依赖于一个项目，它不提供任何对 composer 的支持，你就可以使用这种类型。你基本上就只需要内联一个 composer.json 对象。

```
{
    "repositories": [
        {
            "type": "composer",
            "url": "http://packages.example.com"
        },
        {
            "type": "composer",
            "url": "https://packages.example.com",
            "options": {
                "ssl": {
                    "verify_peer": "true"
                }
            }
        },
        {
            "type": "vcs",
            "url": "https://github.com/Seldaek/monolog"
        },
        {
            "type": "pear",
            "url": "http://pear2.php.net"
        },
        {
            "type": "package",
```

```
        "package": {
            "name": "smarty/smarty",
            "version": "3.1.7",
            "dist": {
                "url": "http://www.smarty.net/files/Smarty-3.1.7.zip",
                "type": "zip"
            },
            "source": {
                "url": "http://smarty-php.googlecode.com/svn/",
                "type": "svn",
                "reference": "tags/Smarty_3_1_7/distribution/"
            }
        }
    }
]
}
```

顺序是非常重要的，当 Composer 查找资源包时，它会按照顺序进行。

默认情况下 Packagist 是最后加入的，因此自定义设置将可以覆盖 Packagist 上的包。

134.1.29 config

下面的这一组选项，仅用于项目。

- `process-timeout`: 默认为 300。处理进程结束时间（例如 git 克隆的时间）。Composer 将放弃超时的任务。如果你的网络缓慢或者正在使用一个巨大的包，你可能要将这个值设置的更高一些。
- `use-include-path`: 默认为 false。如果为 true，Composer autoloader 还将在 PHP include path 中继续查找类文件。
- `preferred-install`: 默认为 auto。它的值可以是 source、dist 或 auto。这个选项允许你设置 Composer 的默认安装方法。
- `github-protocols`: 默认为 `["git", "https", "ssh"]`。从 github.com 克隆时使用的协议优先级清单，因此默认情况下将优先使用 git 协议进行克隆。
  可以重新排列它们的次序，例如如果你的网络有代理服务器或 git 协议的效率很低，你就可以提升 https 协议的优先级。
- `github-oauth`: 一个域名和 oauth keys 的列表。例如，使用 `{"github.com": "oauthtoken"}` 作为此选项的值，将使用 oauthtoken 来访问 github 上的私人仓库，并绕过 low IP-based rate 的 API 限制。
- `vendor-dir`: 默认为 vendor。通过设置你可以安装依赖到不同的目录。
- `bin-dir`: 默认为 vendor/bin。如果一个项目包含二进制文件，它们将被连接到这个目

录。

- `cache-dir`: unix 下默认为 $home/cache，Windows 下默认为 C:\Users\<user>\AppData\Local\Composer。用于存储 composer 所有的缓存文件。
- `cache-files-dir`: 默认为 $cache-dir/files。存储包 zip 存档的目录。
- `cache-repo-dir`: 默认为 $cache-dir/repo。存储 composer 类型的 VCS（svn、github、bitbucket）repos 目录。
- `cache-vcs-dir`: 默认为 $cache-dir/vcs。此目录用于存储 VCS 克隆的 git/hg 类型的元数据，并加快安装速度。
- `cache-files-ttl`: 默认为 15552000（6 个月）。默认情况下 Composer 缓存的所有数据都将在闲置 6 个月后被删除，这个选项允许你来调整这个时间，你可以将其设置为 0 以禁用缓存。
- `cache-files-maxsize`: 默认为 300MiB。Composer 缓存的最大容量，超出后将优先清除旧的缓存数据，直到缓存量低于这个数值。
- `prepend-autoloader`: 默认为 true。如果设置为 false，composer autoloader 将不会附加到现有的自动加载机制中。这有时候用来解决与其它自动加载机制产生的冲突。
- `autoloader-suffix`: 默认为 null。Composer autoloader 的后缀，当设置为空时将会产生一个随机的字符串。optimize-autoloader Defaults to false. Always optimize when dumping the autoloader.
- `github-domains`: 默认为 `["github.com"]`。一个 github mode 下的域名列表。这是用于 GitHub 的企业设置。
- `notify-on-install`: 默认为 true。Composer 允许资源仓库定义一个用于通知的 URL，以便有人从其上安装资源包时能够得到一个反馈通知。此选项允许你禁用该行为。
- `discard-changes`: 默认为 false，它的值可以是 true、false 或 stash。这个选项允许你设置在非交互模式下，当处理失败的更新时采用的处理方式。
  另外，true 表示永远放弃更改，”stash” 表示继续尝试。Use this for CI servers or deploy scripts if you tend to have modified vendors.

```
{
   "config": {
      "bin-dir": "bin"
   }
}
```

134.1.30 scripts

Composer 允许你在安装过程中的各个阶段挂接脚本。

134.1.31 extra

任意的供 scripts 使用的额外数据（可选），这可以是几乎任何东西。若要从脚本事件访问处理程序，你可以这样做：

```
$extra = $event->getComposer()->getPackage()->getExtra();
```

134.1.32 bin

该属性用于标注一组应被视为二进制脚本的文件（可选），它们会被软链接到（config 对象中的）bin-dir 属性所标注的目录，以供其他依赖包调用。

134.1.33 archive

这些选项在创建包存档时使用（可选）。

- exclude: 允许设置一个需要被排除的路径的列表。使用与.gitignore 文件相同的语法。一个前导的（!）将会使其变成白名单而无视之前相同目录的排除设定。前导斜杠只会在项目的相对路径的开头匹配。星号为通配符。

```
{
   "archive": {
      "exclude": ["/foo/bar", "baz", "/*.test", "!/foo/bar/baz"]
   }
}
```

在这个例子中我们 include /dir/foo/bar/file、/foo/bar/baz、/file.php、/foo/my.test 但排除了 /foo/bar/any、/foo/baz、/my.test。

134.2 composer.lock

在安装依赖后，Composer 将把安装时确切的版本号列表写入 composer.lock 文件。

composer.lock 将锁定该项目的特定版本，因此需要提交应用程序的 composer.lock （包括 composer.json）到版本库中

这是非常重要的，因为 install 命令将会检查锁文件是否存在，如果存在就会下载指定的版本（忽略 composer.json 文件中的定义）。

这意味着，任何人建立项目都将下载与指定版本完全相同的依赖。你的持续集成服务器、生产环境、你团队中的其他开发人员、每件事、每个人都使用相同的依赖，从而减轻潜在的错误对部署的影响。即使你独自开发项目，在六个月内重新安装项目时，你也可以放心的继续工作，即使从那时起你的依赖已经发布了许多新的版本。

如果不存在 composer.lock 文件，Composer 将读取 composer.json 并创建锁文件。

这意味着如果你的依赖更新了新的版本，你将不会获得任何更新。此时要更新你的依赖版本请使用 update 命令。这将获取最新匹配的版本（根据你的 composer.json 文件）并将新版本更新进锁文件。

```
$ php composer.phar update
```

如果只想安装或更新一个依赖，你可以白名单它们:

```
$ php composer.phar update monolog/monolog [...]
```

对于库，并不一定建议提交锁文件。如果愿意，可以在自己的项目中提交 composer.lock 文件。

锁文件将帮助团队始终针对同一个依赖版本进行测试，而且这个锁文件任何时候都只对于自己的项目产生影响。

如果不想提交锁文件，并且正在使用 Git，那么需要将它添加到.gitignore 文件中。

134.3 Environment

可以设置一些环境变量来覆盖默认的配置。建议尽可能的在 composer.json 的 config 字段中设置这些值，而不是通过命令行设置环境变量。

值得注意的是，环境变量中的值将始终优先于 composer.json 中所指定的值。

134.3.1 COMPOSER

环境变量 COMPOSER 可以为 composer.json 文件指定其它的文件名。

```
COMPOSER=composer-other.json php composer.phar install
```

134.3.2 COMPOSER_ROOT_VERSION

通过设置这个环境变量，你可以指定 root 包的版本，如果程序不能从 VCS 上猜测出版本号，并且未在 composer.json 文件中申明。

134.3.3 COMPOSER_VENDOR_DIR

通过设置这个环境变量，你可以指定 composer 将依赖安装在 vendor 以外的其它目录中。

134.3.4 COMPOSER_BIN_DIR

通过设置这个环境变量，你可以指定 bin（Vendor Binaries）目录到 vendor/bin 以外的其它目录。

134.3.5 HTTP_PROXY

如果你是通过 HTTP 代理来使用 Composer，你可以使用 http_proxy 或 HTTP_PROXY 环境变量。只要简单的将它设置为代理服务器的 URL。许多操作系统已经为你的服务设置了此变量。

134.3.6 http_proxy

建议使用 http_proxy（小写）或者两者都进行定义。因为某些工具，像 git 或 curl 将使用 http_proxy 小写的版本。另外，你还可以使用 `git config --global http.proxy <proxy url>` 来单独设置 git 的代理。

134.3.7 no_proxy

如果你是使用代理服务器，并且想要对某些域名禁用代理，就可以使用 no_proxy 环境变量。只需要输入一个逗号相隔的域名排除列表。

此环境变量接受域名、IP 以及 CIDR 地址块。你可以将它限制到一个端口（例如:80）。你还可以把它设置为 * 来忽略所有的 HTTP 代理请求。

134.3.8 HTTP_PROXY_REQUEST_FULLURI

如果你使用了 HTTP 代理，但它不支持 request_fulluri 标签，那么你应该设置这个环境变量为 false 或 0 ，来防止 composer 从 request_fulluri 读取配置。

134.3.9 HTTPS_PROXY_REQUEST_FULLURI

如果你使用了 HTTPS 代理，但它不支持 request_fulluri 标签，那么你应该设置这个环境变量为 false 或 0 ，来防止 composer 从 request_fulluri 读取配置。

134.3.10 COMPOSER_HOME

COMPOSER_HOME 环境变量允许你改变 Composer 的主目录。这是一个隐藏的、所有项目共享的全局目录（对本机的所有用户都可用）。

它在各个系统上的默认值分别为：

- *nix
  /home/<user>/.composer
- OSX
  /Users/<user>/.composer

- Windows
  C:\Users\<user>\AppData\Roaming\Composer

134.3.11 COMPOSER_HOME/config.json

你可以在 COMPOSER_HOME 目录中放置一个 config.json 文件。在你执行 install 和 update 命令时，Composer 会将它与你项目中的 composer.json 文件进行合并。

该文件允许你为用户的项目设置配置信息和资源库。

若全局和项目存在相同配置项，那么项目中的 composer.json 文件拥有更高的优先级。

134.3.12 COMPOSER_CACHE_DIR

COMPOSER_CACHE_DIR 环境变量允许你设置 Composer 的缓存目录，这也可以通过 cache-dir 进行配置。

它在各个系统上的默认值分别为:

- *nix
  $COMPOSER_HOME/cache
- OSX
  $COMPOSER_HOME/cache
- Windows
  C:\Users\<user>\AppData\Local\Composer 或%LOCALAPPDATA%/Composer

134.3.13 COMPOSER_PROCESS_TIMEOUT

这个环境变量控制着 Composer 执行命令的等待时间（例如 git 命令）。默认值为 300 秒 (5 分钟)。

134.3.14 COMPOSER_DISCARD_CHANGES

这个环境变量控制着 discard-changes。

134.3.15 COMPOSER_NO_INTERACTION

如果设置为 1，这个环境变量将使 Composer 在执行每一个命令时都放弃交互，相当于对所有命令都使用了`--no-interaction`。可以在搭建虚拟机/持续集成服务器时这样设置。

Chapter 135

Commandline

为了从命令行获得帮助信息，可以运行 composer 或者 composer list 命令，然后结合 `--help` 命令来获得更多的帮助信息。

下列参数可与每一个命令结合使用：

1. `--verbose (-v)`: 增加反馈信息的详细度。
    - `-v` 表示正常输出。
    - `-vv` 表示更详细的输出。
    - `-vvv` 则是为了 debug。
2. `--help (-h)`：显示帮助信息。
3. `--quiet (-q)`：禁止输出任何信息。
4. `--no-interaction (-n)`：不要询问任何交互问题。
5. `--working-dir (-d)`：如果指定的话，使用给定的目录作为工作目录。
6. `--profile`：显示时间和内存使用信息。
7. `--ansi`：强制 ANSI 输出。
8. `--no-ansi`：关闭 ANSI 输出。
9. `--version (-V)`：显示当前应用程序的版本信息。

135.1 init

除了手动创建 composer.json 文件之外，实际上还有一个 init 命令可以更容易的做到这一点。

当运行 `composer init` 命令时，它会以交互方式要求您填写一些信息，同时聪明的使用一些默认值。

```
$ php composer.phar init
```

- `--name`: 包的名称。
- `--description`: 包的描述。
- `--author`: 包的作者。
- `--homepage`: 包的主页。
- `--require`:需要依赖的其它包,必须要有一个版本约束,并且应该遵循 `foo/bar:1.0.0` 这样的格式。
- `--require-dev`: 开发版的依赖包，内容格式与`--require` 相同。
- `--stability (-s)`: minimum-stability 字段的值。

135.2 install

install 命令从当前目录读取 composer.json 文件，处理好依赖关系，并把依赖安装到 vendor 目录下。

```
$ php composer.phar install
```

如果当前目录下存在 composer.lock 文件，它会从此文件读取依赖版本，而不是根据 composer.json 文件去获取依赖，从而确保了该库的每个使用者都能得到相同的依赖版本。

如果没有 composer.lock 文件，composer 将在处理完依赖关系后创建它。

- `--prefer-source`: 下载包的方式有两种：source 和 dist。
  对于稳定版本 composer 将默认使用 dist 方式。而 source 表示版本控制源。如果 `--prefer-source` 是被启用的，composer 将从 source 安装（如果有的话）。如果想要使用一个 bugfix 到你的项目，这是非常有用的。并且可以直接从本地的版本库直接获取依赖关系。
- `--prefer-dist`: 与 `--prefer-source` 相反，composer 将尽可能的从 dist 获取，这将大幅度的加快在 build servers 上的安装。这也是一个回避 git 问题的途径，如果你不清楚如何正确的设置。
- `--dry-run`: 如果你只是想演示而并非实际安装一个包，你可以运行 `--dry-run` 命令，它将模拟安装并显示将会发生什么。
- `--dev`: 安装 require-dev 字段中列出的包（这是一个默认值）。
- `--no-dev`: 跳过 require-dev 字段中列出的包。
- `--no-scripts`: 跳过 composer.json 文件中定义的脚本。
- `--no-plugins`: 关闭 plugins。
- `--no-progress`: 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- `--optimize-autoloader (-o)`: 转换 PSR-0/4 autoloading 到 classmap 可以获得更快的加载支持。特别是在生产环境下建议这么做，但由于运行需要一些时间，因此并没有作为默认值。

135.3 update

为了获取依赖的最新版本，并且升级 composer.lock 文件，可以使用 update 命令。

```
$ php composer.phar update
```

这将解决项目的所有依赖，并将确切的版本号写入 composer.lock。

如果你只是想更新几个包，可以分别列出它们:

```
$ php composer.phar update vendor/package vendor/package2
```

还可以使用通配符进行批量更新:

```
$ php composer.phar update vendor/*
```

- `--prefer-source`: 当有可用的包时，从 source 安装。
- `--prefer-dist`: 当有可用的包时，从 dist 安装。
- `--dry-run`: 模拟命令，并没有做实际的操作。
- `--dev`: 安装 require-dev 字段中列出的包（这是一个默认值）。
- `--no-dev`: 跳过 require-dev 字段中列出的包。
- `--no-scripts`: 跳过 composer.json 文件中定义的脚本。
- `--no-plugins`: 关闭 plugins。
- `--no-progress`: 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- `--optimize-autoloader (-o)`: 转换 PSR-0/4 autoloading 到 classmap 可以获得更快的加载支持。特别是在生产环境下建议这么做，但由于运行需要一些时间，因此并没有作为默认值。
- `--lock`: 仅更新 lock 文件的 hash，取消有关 lock 文件过时的警告。
- `--with-dependencies`: 同时更新白名单内包的依赖关系，这将进行递归更新。

135.4 require

require 命令增加新的依赖包到当前目录的 composer.json 文件中。

```
$ php composer.phar require
```

在添加或改变依赖时，修改后的依赖关系将被安装或者更新。

如果不希望通过交互来指定依赖包，可以在这条指令中直接指明依赖包。

```
$ php composer.phar require vendor/package:2.* vendor/package2:dev-master
```

- `--prefer-source`: 当有可用的包时，从 source 安装。
- `--prefer-dist`: 当有可用的包时，从 dist 安装。

- `--dev`: 安装 require-dev 字段中列出的包。
- `--no-update`: 禁用依赖关系的自动更新。
- `--no-progress`: 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- `--update-with-dependencies`: 一并更新新装包的依赖。

135.5 global

global 命令允许用户在 COMPOSER_HOME 目录下执行其它命令（例如 install、require 或 update 等），并且如果将 $COMPOSER_HOME/vendor/bin 加入到了 $PATH 环境变量中，那么就可以在命令行中安装全局应用。

```
$ php composer.phar global require fabpot/php-cs-fixer:dev-master
```

在 php-cs-fixer 就可以在全局范围使用了（假设已经设置好了 PATH）。如果稍后想更新它，只需要运行 `global update`:

```
$ php composer.phar global update
```

135.6 search

search 命令允许为当前项目搜索依赖包，通常它只搜索 packagist.org 上的包，可以简单的输入搜索条件。

```
$ php composer.phar search monolog
```

也可以通过传递多个参数来进行多条件搜索。

- `--only-name (-N)`: 仅针对指定的名称搜索（完全匹配）。

135.7 show

可以使用 show 命令列出所有可用的软件包。

```
$ php composer.phar show
```

如果想看到一个包的详细信息，可以输入一个包名称。

```
$ php composer.phar monolog/monolog
name    : monolog/monolog
versions : master-dev, 1.0.2, 1.0.1, 1.0.0, 1.0.0-RC1
type    : library
names   : monolog/monolog
```

```
source  : [git] http://github.com/Seldaek/monolog.git 3
    d4e60d0cbc4b888fe5ad223d77964428b1978da
dist    : [zip] http://github.com/Seldaek/monolog/zipball/3
    d4e60d0cbc4b888fe5ad223d77964428b1978da 3d4e60d0cbc4b888fe5ad223d77964428b1978da
license : MIT

autoload
psr-0
Monolog : src/

requires
php >=5.3.0
```

可以输入一个软件包的版本号，来显示该版本的详细信息。

```
$ php composer.phar show monolog 1.0.2
```

- `--installed (-i)`: 列出已安装的依赖包。
- `--platform (-p)`: 仅列出平台软件包（PHP 与它的扩展）。
- `--self (-s)`: 仅列出当前项目信息。

135.8 depends

depends 命令可以查出已安装在你项目中的某个包，是否正在被其它的包所依赖，并列出它们。

```
$ php composer.phar depends --link-type=require monolog/monolog

nrk/monolog-fluent
poc/poc
propel/propel
symfony/monolog-bridge
symfony/symfony
```

- `--link-type`: 检测的类型，默认为 require 也可以是 require-dev。

135.9 validate

在提交 composer.json 文件，和创建 tag 前，应该始终运行 validate 命令。它将检测你的 composer.json 文件是否是有效的。

```
$ php composer.phar validate
```

- `--no-check-all`: Composer 是否进行完整的校验。

135.10 status

如果经常修改依赖包里的代码，并且它们是从 source（自定义源）进行安装的，那么 status 命令允许你进行检查，如果有任何本地的更改它将会给予提示。

```
$ php composer.phar status
```

可以使用`--verbose`系列参数（`-v|vv|vvv`）来获取更详细的详细：

```
$ php composer.phar status -v

You have changes in the following dependencies:
vendor/seld/jsonlint:
   M README.mdown
```

135.11 self-update

将 Composer 自身升级到最新版本，只需要运行 self-update 命令。它将替换你的 composer.phar 文件到最新版本。

```
$ php composer.phar self-update
```

如果想要升级到一个特定的版本，可以这样简单的指定它：

```
$ php composer.phar self-update 1.0.0-alpha7
```

如果已经为整个系统安装 Composer，可能需要在 root 权限下运行它：

```
$ sudo composer self-update
```

- `--rollback (-r)`: 回滚到你已经安装的最后一个版本。
- `--clean-backups`: 在更新过程中删除旧的备份，这使得更新过后的当前版本是唯一可用的备份。

135.12 config

config 命令允许编辑 Composer 的一些基本设置，无论是本地的 composer.json 或者全局的 config.json 文件。

```
$ php composer.phar config --list
config [options] [setting-key] [setting-value1] ... [setting-valueN]
```

setting-key 是一个配置选项的名称，setting-value1 是一个配置的值。可以使用数组作为配置的值（像 github-protocols），多个 setting-value 是允许的。

- `--global (-g)`: 操作位于 $COMPOSER_HOME/config.json 的全局配置文件。如果不指定该参数，此命令将影响当前项目的 composer.json 文件，或 `--file` 参数所指向的文件。
- `--editor (-e)`: 使用文本编辑器打开 composer.json 文件。默认情况下始终是打开当前项目的文件。当存在 `--global` 参数时，将会打开全局 composer.json 文件。
- `--unset`: 移除由 setting-key 指定名称的配置选项。
- `--list (-l)`: 显示当前配置选项的列表。当存在 `--global` 参数时，将会显示全局配置选项的列表。
- `--file="..." (-f)`: 在一个指定的文件上操作，而不是 composer.json。注意，不能与 `--global` 参数一起使用。

除了修改配置选项，config 命令还支持通过以下方法修改来源信息：

```
$ php composer.phar config repositories.foo vcs http://github.com/foo/bar
```

135.13 create-project

可以使用 Composer 从现有的包中创建一个新的项目。这相当于执行了一个 git clone 或 svn checkout 命令后将这个包的依赖安装到它自己的 vendor 目录。

create-project 命令的常见的用途如下：

- 可以快速的部署你的应用。
- 可以检出任何资源包，并开发它的补丁。
- 多人开发项目，可以用它来加快应用的初始化。

要创建基于 Composer 的新项目，可以使用 `create-project` 命令，通过传递一个包名，Composer 会创建项目的目录。

也可以在第三个参数中指定版本号，否则将获取最新的版本。

如果该目录目前不存在，则会在安装过程中自动创建。

```
$ php composer.phar create-project doctrine/orm path 2.2.*
```

此外，也可以无需使用这个命令，而是通过现有的 composer.json 文件来启动这个项目。默认情况下，这个命令会在 packagist.org 上查找指定的包。

- `--repository-url`: 提供一个自定义的储存库来搜索包，这将被用来代替 packagist.org。可以是一个指向 composer 资源库的 HTTP URL，或者是指向某个 packages.json 文件的本地路径。
- `--stability (-s)`: 资源包的最低稳定版本，默认为 stable。
- `--prefer-source`: 当有可用的包时，从 source 安装。
- `--prefer-dist`: 当有可用的包时，从 dist 安装。
- `--dev`: 安装 require-dev 字段中列出的包。

- `--no-install`: 禁止安装包的依赖。
- `--no-plugins`: 禁用 plugins。
- `--no-scripts`: 禁止在根资源包中定义的脚本执行。
- `--no-progress`: 移除进度信息，这可以避免一些不处理换行的终端或脚本出现混乱的显示。
- `--keep-vcs`: 创建时跳过缺失的 VCS 。如果在非交互模式下运行创建命令，这将是非常有用的。

Laravel 框架可以通过如下的命令进行安装：

- 可以全局安装 Laravel 并且在安装完成后可以通过命令新建多个 Laravel 框架

```
$ sudo composer global require "laravel/laravel=~1.1"
```

- 安装一个全新的 Laravel 框架到当前目录

```
$ php composer.phar create-project laravel/laravel --prefer-dist
```

注意，Laravel 需要 PHP 的 openssl 扩展的支持，而且 Laravel 要求 PHP 对 storage 和/bootstrap/cache 目录可写。

```
$ sudo chmod 777 -R storage/
$ sudo chmod 777 bootstrap/cache
```

135.14 dump-autoload

某些情况下需要更新 autoloader（例如在包中加入了一个新的类），可以使用 dump-autoload 来打印自动加载索引，而不必执行 install 或 update 命令。

此外，dump-autoload 可以打印一个优化过的，符合 PSR-0/4 规范的类的索引，这也是出于对性能的可考虑。

在大型的应用中会有许多类文件，而 autoloader 会占用每个请求的很大一部分时间，使用 classmaps 或许在开发时不太方便，但是它在保证性能的前提下，仍然可以获得 PSR-0/4 规范带来的便利。

- `--optimize (-o)`: 转换 PSR-0/4 autoloading 到 classmap 获得更快的载入速度。这特别适用于生产环境，但可能需要一些时间来运行，因此它目前不是默认设置。
- `--no-dev`: 禁用 autoload-dev 规则。

135.15 license

列出已安装的每个包的名称、版本、许可协议。可以使用 `--format=json` 参数来获取 JSON 格式的输出。

135.16 run-script

可以运行此命令来手动执行脚本，只需要指定脚本的名称，可选的 `--no-dev` 参数允许禁用开发者模式。

135.17 diagnose

如果发现了一个 bug 或是程序行为变得怪异，可能需要运行 diagnose 命令来帮助检测一些常见的问题。

```
$ php composer.phar diagnose
```

135.18 archive

archive 命令用来对指定包的指定版本进行 zip/tar 归档，也可以用来归档整个项目（不包括 excluded/ignored（排除/忽略）的文件）。

```
$ php composer.phar archive vendor/package 2.0.21 --format=zip
```

- `--format (-f)`: 指定归档格式：tar 或 zip（默认为 tar）。
- `--dir`: 指定归档存放的目录（默认为当前目录）。

135.19 help

使用 help 可以获取指定命令的帮助信息。

```
$ php composer.phar help install
```

Chapter 136

Autoloading

除了库的下载，Composer 还生成了一个自动加载文件（vendor/autoload.php）。

如果将下面这行代码添加到项目的引导文件中，就可以在需要时自动加载 Composer 下载的库中所有的类文件，例如：

```
require 'vendor/autoload.php';
```

自动加载使得用户可以很容易的使用第三方代码。例如，如果项目依赖 monolog，那么就可以像这样开始使用这个类库，并且他们将被自动加载。

```
$log = new Monolog\Logger('name');
$log->pushHander(new Monolog\Handler\StreamHandler('app.log',Monolog\Logger::WARNING));

$log->addWarning('Foo');
```

可以在 composer.json 的 autoload 字段中增加自己的 autoloader。例如，下面的说明将指示 Composer 注册一个 PSR-4 autoloader 到 Acme 命名空间。

```
{
  "autoload": {
    "psr-4": {"Acme\\": "src/"}
  }
}
```

可以定义一个从命名空间到目录的映射。此时 src 会在项目的根目录，与 vendor 文件夹同级。例如 src/Foo.php 文件应该包含 Acme\Foo 类。

添加 autoload 字段后，应该再次运行 install 命令来生成 vendor/autoload.php 文件。

引用这个文件也将返回 autoloader 的实例，可以将包含调用的返回值存储在变量中，并添加更多的命名空间。这个事实对于在一个测试套件中自动加载类文件是非常有用的，例如：

```
$loader = require 'vendor/autoload.php';
$loader->add('Acme\\Test\\',__DIR__);
```

除了 PSR-4 自动加载，classmap 也是支持的，这样就允许类被自动加载（即使不符合 PSR-0 规范）。

Composer 提供了自己的 autoloader。如果不想使用它，可以仅仅引入 vendor/composer/autoload_*.php 文件，它返回一个关联数组，可以通过这个关联数组配置自己的 autoloader。

136.1 Spec

136.1.1 PSR-0

136.1.2 PSR-4

136.2 Init

在服务器的 web 目录下创建一个 lara 目录作为网站的根目录，并且在该目录下创建一个 composer.json 文件，例如：

```
$ cd /srv/web
$ mkdir lara
$ cd lara
$ cat > composer.json
{
   "require": {

   }
}
```

接下来的 lara 目录下执行 `composer update` 来开始项目初始化，完成后在 lara 目录下会自动生成自动加载文件，例如：

```
$ cd /srv/web/lara
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
Nothing to install or update
Generating autoload files
```

```
$ tree /srv/web/lara
/srv/web/lara
|— composer.json
```

```
|— vendor
   |— autoload.php
   |— composer
        |— autoload_classmap.php
        |— autoload_namespaces.php
        |— autoload_psr4.php
        |— autoload_real.php
        |— autoload_static.php
        |— ClassLoader.php
        |— installed.json
        |— LICENSE
```

136.3 Route

在项目初始化完成后，首先添加的第一个组件就是路由组件。

这里选择使用 illuminate/routing 作为路由组件，该路由组件同时也通过自己的 composer.json 文件依赖其他组件（例如 symfony/routing 和 illuminate/container。

在添加 illuminate/routing 组件的同时还需要添加 illuminate/events 组件。

```
$ vim /srv/web/lara
{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*"
  }
}
$ cd /srv/web/lara
$ composer update
```

在添加路由组件成功，接下来首先需要添加两个文件，一个是路由文件，另一个是服务器端程序入口文件。

```
$ tree /srv/web/lara
/srv/web/lara
|— app
|       |— Http
|             |— routes.php
|— composer.json
|— vendor
   |— autoload.php
   |— composer
```

```
        |— autoload_classmap.php
        |— autoload_namespaces.php
        |— autoload_psr4.php
        |— autoload_real.php
        |— autoload_static.php
        |— ClassLoader.php
        |— installed.json
        |— LICENSE
```

136.3.1 Router

对于路由配置文件，需要在 lara 目录下创建一个 app 目录，然后在 app 目录下创建 Http 目录来存储处理 HTTP 请求的文件。

路由文件放置在 Http 目录下，路由文件的代码如下：

```
<?php
$app['router']->get('/', function () {
   return '<h1>路由成功</h1>';
});
```

在路由文件中可以通过路由实例的对应方法（例如 get()）添加路由规则，例如/表示网站根目录。

路由的处理函数是一个匿名函数，可以用来返回响应。

具体来说，入口文件通过 Illuminate\Http\Request 类的静态方法 createFromGlobals() 实现请求的实例化，然后通过路由进行分发处理。

1. 路由根据请求的地址查找路由表。
2. 从路由表中找到对应的函数来处理请求并返回响应，如果从路由表无法找到对应的处理函数就返回 404 错误。

以上就是路由的基本过程。

136.3.2 Entry

服务器端程序的请求入口文件位于 lara 目录下的 public 目录中。

public 目录用于存放项目的公共文件，也就是通过 HTTP 请求可以访问到的文件（包括入口文件、js 文件和 CSS 文件等）。

```
$ tree /srv/web/lara
/srv/web/lara
|— app
|       |— Http
|              |— routes.php
```

```
|— public
|       |— index.php
|— composer.json
|— vendor
   |— autoload.php
   |— composer
       |— autoload_classmap.php
       |— autoload_namespaces.php
       |— autoload_psr4.php
       |— autoload_real.php
       |— autoload_static.php
       |— ClassLoader.php
       |— installed.json
       |— LICENSE
```

在/srv/web/lara/public 目录下创建一个 index.php 文件作为服务器端程序的请求入口文件。

```
<?php
// 调用自动加载文件，添加自动加载文件函数
require __DIR__ . '/../vendor/autoload.php';

// 实例化服务容器，注册事件、路由服务提供者
$app = new Illuminate\Container\Container();
with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();

// 加载路由
require __DIR__ . '/../app/Http/routes.php';

// 实例化请求并分发处理请求
$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
// 返回请求响应
$response->send();
```

入口文件完成的主要工作如下：

- 添加自动加载函数
- 服务容器实例化和服务注册
- 路由加载
- 请求实例化和路由分发
- 响应生成与发送

自动加载函数用于自动包含引用文件，需要首先添加。

136.4 Container

Laravel 框架的一些功能的生成都需要服务容器来实现，即 Illuminate\Container\Container 类的实例。

具体来说，服务容器用于服务注册和解析，也就是说向服务容器注册能够实现某些功能的实例或回调函数，这样当需要某些功能（例如事件和路由功能）时就可以从服务容器中获取相应的实例来完成。

在实例化服务容器后立刻注册了事件和路由的服务，然后就可以通过这个服务容器（$app）来获取路由处理的相关实例并完成路由的加载和请求处理。

其中，获取路由实例通过 `$app['router']` 实现。

- `$app`：服务容器实例
- `['router']`：服务容器中注册的路由服务名称

```
<?php
var_dump($app);

object(Illuminate\Container\Container)[3]
  protected 'resolved' =>
    array (size=0)
      empty
  protected 'bindings' =>
    array (size=7)
      'events' =>
        array (size=2)
          'concrete' =>
            object(Closure)[4]
              ...
          'shared' => boolean true
      'router' =>
        array (size=2)
          'concrete' =>
            object(Closure)[7]
              ...
          'shared' => boolean false
      'url' =>
        array (size=2)
          'concrete' =>
            object(Closure)[9]
```

```
          ...
       'shared' => boolean false
    'redirect' =>
      array (size=2)
       'concrete' =>
         object(Closure)[11]
           ...
       'shared' => boolean false
    'Psr\Http\Message\ServerRequestInterface' =>
      array (size=2)
       'concrete' =>
         object(Closure)[12]
           ...
       'shared' => boolean false
    'Psr\Http\Message\ResponseInterface' =>
      array (size=2)
       'concrete' =>
         object(Closure)[13]
           ...
       'shared' => boolean false
    'Illuminate\Contracts\Routing\ResponseFactory' =>
      array (size=2)
       'concrete' =>
         object(Closure)[14]
           ...
       'shared' => boolean true
protected 'instances' =>
  array (size=0)
    empty
protected 'aliases' =>
  array (size=0)
    empty
protected 'extenders' =>
  array (size=0)
    empty
protected 'tags' =>
  array (size=0)
    empty
protected 'buildStack' =>
  array (size=0)
    empty
```

```
  public 'contextual' =>
    array (size=0)
      empty
  protected 'reboundCallbacks' =>
    array (size=0)
      empty
  protected 'globalResolvingCallbacks' =>
    array (size=0)
      empty
  protected 'globalAfterResolvingCallbacks' =>
    array (size=0)
      empty
  protected 'resolvingCallbacks' =>
    array (size=0)
      empty
  protected 'afterResolvingCallbacks' =>
    array (size=0)
      empty
```

136.5 Request

```
<?php
var_dump($request);

object(Illuminate\Http\Request)[22]
  protected 'json' => null
  protected 'convertedFiles' => null
  protected 'userResolver' => null
  protected 'routeResolver' => null
  public 'attributes' =>
    object(Symfony\Component\HttpFoundation\ParameterBag)[25]
      protected 'parameters' =>
        array (size=0)
          empty
  public 'request' =>
    object(Symfony\Component\HttpFoundation\ParameterBag)[23]
      protected 'parameters' =>
        array (size=0)
          empty
  public 'query' =>
```

```
  object(Symfony\Component\HttpFoundation\ParameterBag)[24]
    protected 'parameters' =>
      array (size=0)
        empty
public 'server' =>
  object(Symfony\Component\HttpFoundation\ServerBag)[28]
    protected 'parameters' =>
      array (size=34)
        'USER' => string 'www-data' (length=8)
        'HOME' => string '/var/www' (length=8)
        'HTTP_ACCEPT_LANGUAGE' => string 'en-US,en;q=0.8,zh-CN;q=0.6,zh;q=0.4,zh-TW;q
            =0.2' (length=47)
        'HTTP_ACCEPT_ENCODING' => string 'gzip, deflate, sdch' (length=19)
        'HTTP_DNT' => string '1' (length=1)
        'HTTP_ACCEPT' => string 'text/html,application/xhtml+xml,application/xml;q=0.9,
            image/webp,*/*;q=0.8' (length=74)
        'HTTP_USER_AGENT' => string 'Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
             (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36' (length=104)
        'HTTP_UPGRADE_INSECURE_REQUESTS' => string '1' (length=1)
        'HTTP_CACHE_CONTROL' => string 'max-age=0' (length=9)
        'HTTP_CONNECTION' => string 'keep-alive' (length=10)
        'HTTP_HOST' => string 'lara.in' (length=7)
        'REDIRECT_STATUS' => string '200' (length=3)
        'SERVER_NAME' => string 'lara.in' (length=7)
        'SERVER_PORT' => string '80' (length=2)
        'SERVER_ADDR' => string '172.17.0.18' (length=11)
        'REMOTE_PORT' => string '47842' (length=5)
        'REMOTE_ADDR' => string '172.17.0.1' (length=10)
        'PATH_INFO' => string '' (length=0)
        'SCRIPT_NAME' => string '/index.php' (length=10)
        'SERVER_SOFTWARE' => string 'nginx' (length=5)
        'GATEWAY_INTERFACE' => string 'CGI/1.1' (length=7)
        'SERVER_PROTOCOL' => string 'HTTP/1.1' (length=8)
        'DOCUMENT_ROOT' => string '/usr/share/nginx/lara/public' (length=28)
        'DOCUMENT_URI' => string '/index.php' (length=10)
        'REQUEST_URI' => string '/' (length=1)
        'CONTENT_LENGTH' => string '' (length=0)
        'CONTENT_TYPE' => string '' (length=0)
        'REQUEST_METHOD' => string 'GET' (length=3)
        'QUERY_STRING' => string '' (length=0)
```

```
        'SCRIPT_FILENAME' => string '/usr/share/nginx/lara/public/index.php' (length
            =38)
        'FCGI_ROLE' => string 'RESPONDER' (length=9)
        'PHP_SELF' => string '/index.php' (length=10)
        'REQUEST_TIME_FLOAT' => float 1481011635.6761
        'REQUEST_TIME' => int 1481011635
public 'files' =>
  object(Symfony\Component\HttpFoundation\FileBag)[27]
    protected 'parameters' =>
      array (size=0)
        empty
public 'cookies' =>
  object(Symfony\Component\HttpFoundation\ParameterBag)[26]
    protected 'parameters' =>
      array (size=0)
        empty
public 'headers' =>
  object(Symfony\Component\HttpFoundation\HeaderBag)[29]
    protected 'headers' =>
      array (size=11)
        'accept-language' =>
          array (size=1)
            ...
        'accept-encoding' =>
          array (size=1)
            ...
        'dnt' =>
          array (size=1)
            ...
        'accept' =>
          array (size=1)
            ...
        'user-agent' =>
          array (size=1)
            ...
        'upgrade-insecure-requests' =>
          array (size=1)
            ...
        'cache-control' =>
          array (size=1)
            ...
```

```
        'connection' =>
          array (size=1)
            ...
        'host' =>
          array (size=1)
            ...
        'content-length' =>
          array (size=1)
            ...
        'content-type' =>
          array (size=1)
            ...
    protected 'cacheControl' =>
      array (size=1)
        'max-age' => string '0' (length=1)
  protected 'content' => null
  protected 'languages' => null
  protected 'charsets' => null
  protected 'encodings' => null
  protected 'acceptableContentTypes' => null
  protected 'pathInfo' => null
  protected 'requestUri' => null
  protected 'baseUrl' => null
  protected 'basePath' => null
  protected 'method' => null
  protected 'format' => null
  protected 'session' => null
  protected 'locale' => null
  protected 'defaultLocale' => string 'en' (length=2)
```

136.6 Response

```
<?php
var_dump($response);

object(Illuminate\Http\Response)[39]
  public 'original' => string '<h1>路由成功</h1>' (length=21)
  public 'headers' =>
    object(Symfony\Component\HttpFoundation\ResponseHeaderBag)[40]
      protected 'computedCacheControl' =>
```

```
      array (size=1)
        'no-cache' => boolean true
    protected 'cookies' =>
      array (size=0)
        empty
    protected 'headerNames' =>
      array (size=2)
        'cache-control' => string 'Cache-Control' (length=13)
        'content-type' => string 'Content-Type' (length=12)
    protected 'headers' =>
      array (size=2)
        'cache-control' =>
          array (size=1)
            ...
        'content-type' =>
          array (size=1)
            ...
    protected 'cacheControl' =>
      array (size=0)
        empty
protected 'content' => string '<h1>路由成功</h1>' (length=21)
protected 'version' => string '1.1' (length=3)
protected 'statusCode' => int 200
protected 'statusText' => string 'OK' (length=2)
protected 'charset' => null
public 'exception' => null
```

136.7 Controller

实际项目中的请求处理往往比较复杂，这样就不便于把请求的处理函数直接写在路由文件中。

一般情况下，路由的请求处理部分单独使用控制器类来实现，而且在添加路由组件时已经添加了基本控制器类，即 Illuminate\Routing\Controller 类，因此在添加控制器模块时可以使用这个类作为基类对控制器的功能进行扩展。

如果需要让不同模块之间功能划分清晰，也可以不使用 Illuminate\Routing\Controller 类作为基类，而是直接创建控制器类。

在/srv/web/lara/app/Http 目录下创建 Controllers 目录，开始添加第一个 Controller 类——WelcomeController.php。

```
$ tree /srv/web/lara
```

```
/srv/web/lara
|— app
|       |— Http
|              |— Controllers
|              |      |— WelcomeController.php
|              |— routes.php
|— public
|       |— index.php
|— composer.json
|— vendor
   |— autoload.php
   |— composer
       |— autoload_classmap.php
       |— autoload_namespaces.php
       |— autoload_psr4.php
       |— autoload_real.php
       |— autoload_static.php
       |— ClassLoader.php
       |— installed.json
       |— LICENSE
```

为了实现文件的自动加载，还需要根据 PSR-4 规范进行相关配置，并在 composer.json 中添加自动加载路径，然后执行 `composer dump-autoload` 更新自动加载文件。

```
{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app/"
    }
  }
}
```

在浏览器中访问 lara.in/welcome 时可以打印出 `PATH_INFO` 如下：

```
'PATH_INFO' => '/welcome'
```

对于控制器类文件，需要按照文件路径添加正确的命名空间，其中类名要与文件名相同，@ 后面跟的是控制器类的处理函数的名称。

136.8 Model

除了可以在控制器类中添加相应的组件之外，还可以继续添加模型组件（相当于 MVC 模式中的 M）来实现数据处理功能。

在添加模型组件之前，需要修改 composer.json 文件并更新来最终完成模型组件及其依赖的下载。

```
{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*",
    "illuminate/database": "*"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app/"
    }
  }
}
```

illuminate\database 组件主要用于操作数据库，而且该组件提供了两种操作数据库的方式，分别是查询构造器方式和 Eloquent ORM 方式。

136.8.1 Eloquent ORM

Eloquent ORM 方式操作数据库需要完成五个步骤，分别是创建数据库、添加数据库配置信息、启动 Eloquent ORM 模块、创建 model 类和通过 model 类来操作数据库。

```
$ tree /srv/web/lara
/srv/web/lara
|— app
|       |— Http
|              |— Controllers
|              |      |— WelcomeController.php
|              |— routes.php
|— config
|       |— api.php
|       |— app.php
|       |— cache.php
|       |— database.php
|       |— mail.php
|       |— oauth2.php
```

```
|       |— queue.php
|       |— services.php
|       |— session.php
|       |— view.php
|— public
|       |— index.php
|— composer.json
|— vendor
    |— autoload.php
    |— composer
        |— autoload_classmap.php
        |— autoload_namespaces.php
        |— autoload_psr4.php
        |— autoload_real.php
        |— autoload_static.php
        |— ClassLoader.php
        |— installed.json
        |— LICENSE
```

按照 Laravel 框架的目录结构，在/srv/web/lara 目录下创建 config 目录来保存整个应用程序的配置文件。

- database.php 存储对数据库的配置信息（以数组的形式）
- api.php 存储 API 接口的配置信息（例如接口频率限制等）
- app.php 存储应用程序的配置信息（例如时区信息等）
- cache.php 存储缓存数据库的配置信息（例如 Memcached 和 Redis 等）
- mail.php 存储邮件服务器的配置信息
- oauth2.php 存储使用第三方登录的 OAuth2 配置信息
- queue.php 存储队列服务器信息（例如 Redis 等）
- services.php 存储接入的第三方服务（例如支付、微信等）
- session.php 存储会话存储的配置信息
- view.php 存储视图的配置信息

```
$ cat > /srv/web/lara/config/database.php
<?php
return [
   'driver'=>'mysql',
   'host'=>'localhost',
   'database'=>'lara',
   'username'=>'root',
   'password'=>'rootp@ssw0rd',
   'charset'=>'utf8',
```

```
  'collation'=>'utf8_general_ci',
  'prefix'=>'lara_'
];
```

在配置数据库后就可以在访问入口文件中启动 Eloquent ORM 模块。

```
$ cat > /srv/web/lara/public/index.php
use Illuminate\Database\Capsule\Manager;
require __DIR__ . '/../vendor/autoload.php';

$app = new Illuminate\Container\Container();

with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();
$manager = new Manager();
$manager->addConnection(require '../config/database.php');
$manager->bootEloquent();

require __DIR__ . '/../app/Http/routes.php';

$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
$response->send();
```

启动 Eloquent ORM 模块需要使用数据库的管理类，即 Illuminate\Database\Capsule\Manager 类，因此需要添加对应的命名空间并进行初始化。

- addConnection() 函数完成数据库的相关配置
- bootEloquent() 函数完成 Eloquent ORM 模块的启动

启动 Eloquent ORM 模块后就可以操作数据库，注意 Eloquent ORM 操作数据库需要两个步骤来实现，分别是创建模型类和通过模型类的方法操作数据库。

```
$ tree /srv/web/lara
/srv/web/lara
|— app
|       |— Http
|       |       |— Controllers
|       |       |       |— WelcomeController.php
|       |       |— routes.php
|       |— Models
|               |— Student.php
|— config
|       |— api.php
|       |— app.php
```

```
|       |— cache.php
|       |— database.php
|       |— mail.php
|       |— oauth2.php
|       |— queue.php
|       |— services.php
|       |— session.php
|       |— view.php
|— public
|       |— index.php
|— composer.json
|— vendor
    |— autoload.php
    |— composer
        |— autoload_classmap.php
        |— autoload_namespaces.php
        |— autoload_psr4.php
        |— autoload_real.php
        |— autoload_static.php
        |— ClassLoader.php
        |— installed.json
        |— LICENSE
```

模型类可以保存在 app 目录下的 Models 目录下来统一管理。例如，建立/srv/web/lara/app/Models/Student.php 来创建 App\Models\Student 模型类如下：

```
$ cat > /srv/web/lara/app/Models/Student.php
<?php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Student extends Model {
    public $timestamps = false;
}
```

Eloquent ORM 操作数据库时首先需要引入模型类，例如使用 use 关键字引入 App\Models\Student 模型类，接下来就可以使用 first() 函数获取 students 表中的第一行数据（封装在模型类实例 $student 中）。

模型类实例的 getAttributes() 函数可以返回相应的数据并展示在页面上。

- 每个模型类都需要继承 Illuminate\Database\Eloquent\Model 类。
- 每个模型类只对应一个数据表（默认情况下类名小写后的复数就是对应的表名）。

在通过模型类调用相应方法时就会操作对应的表中的数据（例如 first() 函数可以获取表中的第一行数据）。

136.9 View

Laravel 框架的 Illuminate\view 视图组件可以将视图以模板的方式创建，而且在其他视图中也可以调用、继承已经创建的模板，并通过模板语法来简化视图设计。

更新 composer.json 文件来引入 Illuminate\view 视图组件。

```
{
  "require": {
    "illuminate/routing": "*",
    "illuminate/events": "*",
    "illuminate/database": "*",
    "illuminate/view": "*"
  },
  "autoload": {
    "psr-4": {
      "App\\": "app/"
    }
  }
}
```

视图组件的使用需要完成以下 4 个工作，分别是：

1. 添加视图模板文件和编译文件的存储路径；
2. 对视图进行相关配置和服务注册；
3. 使用视图文件；
4. 创建视图模板文件。

Illuminate\view 视图组件将视图模板文件编译成普通的视图文件并进行存储，因此第一步需要添加相应的存储路径。

```
$ tree /srv/web/lara
/srv/web/lara
|— app
|       |— Http
|       |       |— Controllers
|       |       |      |— WelcomeController.php
|       |       |— routes.php
|       |— Models
|               |— Student.php
|— config
```

```
|       |— api.php
|       |— app.php
|       |— cache.php
|       |— database.php
|       |— mail.php
|       |— oauth2.php
|       |— queue.php
|       |— services.php
|       |— session.php
|       |— view.php
|— public
|       |— index.php
|— resources
|       |— views
|               |— welcome.blade.php
|— storage
|       |— framework
|               |— views
|                       |— 7f8ac86c71cd9a8c22ddf0a79647c4e238965f6a.php
|— composer.json
|— vendor
   |— autoload.php
   |— composer
       |— autoload_classmap.php
       |— autoload_namespaces.php
       |— autoload_psr4.php
       |— autoload_real.php
       |— autoload_static.php
       |— ClassLoader.php
       |— installed.json
       |— LICENSE
```

默认情况下，视图模板文件存储在 resource/views 目录下，编译出的视图文件存储在 storage/framework/views 目录下，其中 storage 目录和 bootstrap/cache 目录需要设置为 PHP 可写权限，例如：

```
$ sudo chmod 777 -R storage/
```

在 index.php 文件中完成视图组件的相关配置和服务注册如下：

```
<?php
use Illuminate\Database\Capsule\Manager;
use Illuminate\Support\Fluent;
```

```
require __DIR__ . '/../vendor/autoload.php';

$app = new Illuminate\Container\Container();
Illuminate\Container\Container::setInstance($app);
with(new Illuminate\Events\EventServiceProvider($app))->register();
with(new Illuminate\Routing\RoutingServiceProvider($app))->register();
// Eloquent ORM
$manager = new Manager();
$manager->addConnection(require '../config/database.php');
$manager->bootEloquent();

$app->instance('config',new Fluent);
$app['config']['view.compiled']='../storage/framework/views';
$app['config']['view.paths']=['../resources/views'];
with(new Illuminate\View\ViewServiceProvider($app))->register();
with(new Illuminate\Filesystem\FilesystemServiceProvider($app))->register();
require __DIR__ . '/../app/Http/routes.php';

$request = Illuminate\Http\Request::createFromGlobals();
$response = $app['router']->dispatch($request);
$response->send();
```

136.9.1 setInstance()

服务容器的 setInstance() 静态方法可以将服务容器实例添加为静态属性，从而可以在任何位置获取服务容器的实例。

136.9.2 instance()

在配置视图模块时，首先通过服务容器实例的 instance() 方法将服务名称为 config 和 Illuminate\Support\Fluent 类的实例进行绑定。

Illuminate\Support\Fluent 类的实例主要用于存储视图模块的配置信息，这里的配置信息包括视图模板文件和编译文件的存储路径，需要分别添加到配置实例中。

视图模块的使用需要文件模块的支持，在下载视图组件时，文件模块会作为依赖下载，因此可以直接使用文件组件的服务提供者进行服务注册。

在使用视图组件时，在处理函数中使用视图组件实现视图的加载，其中路由文件和在控制器模块中添加的相同，这里还是使用 index() 函数进行处理。

视图模块返回视图响应的处理函数内容如下：

```
<?php
```

```
namespace App\Http\Controllers;

use App\Models\Student;
use Illuminate\Container\Container;

class WelcomeController{
    public function index(){
        $student = Student::first();
        $data = $student->getAttributes();
        $app = Container::getInstance();
        $factory = $app->make('view');
        return $factory->make('welcome')->with('data',$data);
    }
}
```

- 服务容器的 getInstance() 静态方法获取到服务容器的实例。
- 服务容器获取服务名称为 view 的实例对象(即视图创建工厂类(Illuminate\View\Factory)的实例)，其中的参数就是视图文件的名称，实际上是在视图模板文件路径中查找对应文件名的文件，视图文件就在后面创建。
- 视图实例的 with() 方法用来添加数据，这样就可以在视图文件中使用。

视图模板文件默认位于/resources/views 目录下 (例如 welcome.blade.php), Illuminate\view 视图组件规定模板文件使用.blade.php 为后缀，而且文件名要和视图创建工厂的 make() 方法中的字符串参数相同。

```
$ cat > welcome.blade.php
<h3>Students</h3>
<p>id:{{$data['id']}};</p>
<p>name:{{$data['name']}};</p>
<p>age:{{$data['age']}};</p>
```

在视图文件中可以按照 HTML 格式创建视图，传入的数据可以通过 `{{变量}}` 的形式输出。

Part XXX

PSR

Chapter 137

Overview

PSR 是 PHP Standard Recommendations 的简写，由 PHP FIG 组织制定的 PHP 规范，是 PHP 开发的实践标准。

用户可以使用 PHP_CodeSniffer 来检查代码是否符合 PSR 代码规范。例如，可以通过这些工具来实时地自动修正程序语法使其符合标准。

- PHP Coding Standards Fixer
- php.tools（sublime-phpfmt）

也可以手动运行 phpcs 命令来输出相应的错误以及如何修正的方法。同样地，phpcs 也可以用在 git hook 中，这样如果分支代码不符合选择的代码标准则无法提交。

```
$ phpcs -sw --standard=PSR2 file.php
```

137.1 Introduction

FIG 是 Framework Interoperability Group（框架可互用性小组）的缩写，由几位开源框架的开发者成立于 2009 年，从那开始也选取了很多其他成员进来（包括但不限于 Laravel, Joomla, Drupal, Composer, Phalcon, Slim, Symfony, Zend Framework 等），虽然不是「官方」组织，但也代表了大部分的 PHP 社区。

FIG 项目的目的在于：通过框架作者或者框架的代表之间讨论，以最低程度的限制，制定一个协作标准，各个框架遵循统一的编码规范，避免各家自行发展的风格阻碍了 PHP 的发展，解决这个程序设计师由来已久的困扰。

目前已表决通过了的 6 套 PSR 标准，废弃了 1 套标准（自动加载规范），已经得到大部分 PHP 框架的支持和认可。

- 基础编码规范
- 编码风格规范

- 日志接口规范
- 自动加载规范
- 缓存接口规范
- HTTP 消息接口规范

正在起草中的规范包括：

- PHPDoc 标准
- Huggable 接口
- 项目安全问题公示
- 项目安全上报方法
- 服务容器接口
- 全量编码风格规范
- 超媒体链接

Chapter 138

PSR-1

138.1 Overview

- PHP 代码文件必须以 <?php 或 <?= 标签开始;
- PHP 代码文件必须以不带 BOM 的 UTF-8 编码;
- PHP 代码中应该只定义类、函数、常量等声明，或其他会产生副作用的操作（例如生成文件输出以及修改.ini 配置文件等)，二者只能选其一;
- 命名空间以及类必须符合 PSR 的自动加载规范：PSR-4 中的一个;
- 类的命名必须遵循 StudlyCaps 大写开头的驼峰命名规范;
- 类中的常量所有字母都必须大写，单词间用下划线分隔;
- 方法名称必须符合 camelCase 式的小写开头驼峰命名规范。

138.2 File

138.2.1 PHP Tag

PHP 代码必须使用 <?php ?> 长标签或 <?= ?> 短输出标签；一定不可使用其它自定义标签。

138.2.2 PHP Charset

PHP 代码必须且只可使用不带 BOM 的 UTF-8 编码。

138.2.3 Side Effect

一份 PHP 文件中应该要不就只定义新的声明，如类、函数或常量等不产生副作用的操作，要不就只书写会产生副作用的逻辑操作，但不该同时具有两者。

「副作用」(side effects) 一词的意思是，仅仅通过包含文件，不直接声明类、函数和常量等，而执行的逻辑操作。

「副作用」包含却不仅限于：

- 生成输出
- 直接的 require 或 include
- 连接外部服务
- 修改 ini 配置
- 抛出错误或异常
- 修改全局或静态变量
- 读或写文件等

以下是一个反例，一份包含「函数声明」以及产生「副作用」的代码：

```
<?php
// 「副作用」：修改 ini 配置
ini_set('error_reporting', E_ALL);

// 「副作用」：引入文件
include "file.php";

// 「副作用」：生成输出
echo "<html>\n";

// 声明函数
function foo()
{
    // 函数主体部分
}
```

下面是一个范例，一份只包含声明不产生「副作用」的代码：

```
<?php
// 声明函数
function foo()
{
    // 函数主体部分
}

// 条件声明 **不** 属于「副作用」
```

```
if (! function_exists('bar')) {
    function bar()
    {
        // 函数主体部分
    }
}
```

138.3 Namespace

命名空间以及类的命名必须遵循 PSR-4。

根据规范，每个类都独立为一个文件，且命名空间至少有一个层次：顶级的组织名称（vendor name）。

类的命名必须遵循 StudlyCaps 大写开头的驼峰命名规范。

PHP 5.3 及以后版本的代码必须使用正式的命名空间。

```
<?php
// PHP 5.3及以后版本的写法
namespace Vendor\Model;

class Foo
{
}
```

5.2.x 及之前的版本应该使用伪命名空间的写法，约定俗成使用顶级的组织名称（vendor name）如 Vendor_ 为类前缀。

```
<?php
// 5.2.x及之前版本的写法
class Vendor_Model_Foo
{
}
```

138.4 Class Member

此处的「类」指代所有的类、接口以及可复用代码块（traits）。

138.4.1 Constants

类的常量中所有字母都必须大写，词间以下划线分隔。

```
<?php
namespace Vendor\Model;

class Foo
{
    const VERSION = '1.0';
    const DATE_APPROVED = '2012-06-01';
}
```

138.4.2 Property

类的属性命名可以遵循:

- 大写开头的驼峰式 ($StudlyCaps)
- 小写开头的驼峰式 ($camelCase)
- 下划线分隔式 ($under_score)

本规范不做强制要求，但无论遵循哪种命名方式，都应该在一定的范围内保持一致。这个范围可以是整个团队、整个包、整个类或整个方法。

138.4.3 Method

方法名称必须符合 camelCase() 式的小写开头驼峰命名规范。

Chapter 139

PSR-2

PSR-2 的代码必须符合 PSR-1 中的所有规范。

139.1 Overview

- 代码必须遵循 PSR-1 中的编码规范。
- 代码必须使用 4 个空格符而不是「Tab 键」进行缩进。
- 每行的字符数应该软性保持在 80 个之内，理论上一定不可多于 120 个，但一定不可有硬性限制。
- 每个 namespace 命名空间声明语句和 use 声明语句块后面，必须插入一个空白行。
- 类的开始花括号（{）必须写在函数声明后自成一行，结束花括号（}）也必须写在函数主体后自成一行。
- 方法的开始花括号（{）必须写在函数声明后自成一行，结束花括号（}）也必须写在函数主体后自成一行。
- 类的属性和方法必须添加访问修饰符（private、protected 以及 public），abstract 以及 final 必须声明在访问修饰符之前，而 static 必须声明在访问修饰符之后。
- 控制结构的关键字后必须要有一个空格符，而调用方法或函数时则一定不可有。
- 控制结构的开始花括号（{）必须写在声明的同一行，而结束花括号（}）必须写在主体后自成一行。
- 控制结构的开始左括号后和结束右括号前，都一定不可有空格符。

```
<?php
namespace Vendor\Package;

use FooInterface;
use BarClass as Bar;
```

```
use OtherVendor\OtherPackage\BazClass;

class Foo extends Bar implements FooInterface
{
    public function sampleFunction($a, $b = null)
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // 方法的内容
    }
}
```

139.2 File

- 所有 PHP 文件必须使用 Unix LF (linefeed) 作为行的结束符。
- 所有 PHP 文件必须以一个空白行作为结束。
- 纯 PHP 代码文件必须省略最后的 ?> 结束标签。

139.3 Line

- 行的长度一定不可有硬性的约束。
- 软性的长度约束必须要限制在 120 个字符以内，若超过此长度，带代码规范检查的编辑器必须要发出警告，不过一定不可发出错误提示。
- 每行不该多于 80 个字符，大于 80 字符的行应该折成多行。
- 非空行后一定不可有多余的空格符。
- 空行可以使得阅读代码更加方便以及有助于代码的分块。
- 每行一定不可存在多于一条语句。

139.4 Indent

代码必须使用 4 个空格符的缩进，一定不可用 tab 键。

使用空格而不是「tab 键缩进」的好处在于，避免在比较代码差异、打补丁、重阅代码以及注释时产生混淆。并且，使用空格缩进，让对齐变得更方便。

139.5 Keyword

- PHP 所有关键字必须全部小写。
- 常量 true 、 false 和 null 也必须全部小写。

139.6 Namespace

- namespace 声明后必须插入一个空白行。
- 所有 use 必须在 namespace 后声明。
- 每条 use 声明语句必须只有一个 use 关键词。
- use 声明语句块后必须要有一个空白行。

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

// ... 更多的 PHP 代码在这里 ...
```

139.7 Class Member

此处的「类」泛指所有的「class 类」、「接口」以及「traits 可复用代码块」。

139.7.1 Extend

关键词 extends 和 implements 必须写在类名称的同一行。

类的开始花括号必须独占一行，结束花括号也必须在类主体后独占一行。

```
<?php
namespace Vendor\Package;
```

```
use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // 这里面是常量、属性、类方法
}
```

139.7.2 Inherit

implements 的继承列表也可以分成多行，这样的话，每个继承接口名称都必须分开独立成行，包括第一个。

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // 这里面是常量、属性、类方法
}
```

139.7.3 Property

- 每个属性都必须添加访问修饰符。
- 一定不可使用关键字 var 声明一个属性。
- 每条语句一定不可定义超过一个属性。
- 不该使用下划线作为前缀，来区分属性是 protected 或 private。

```
<?php
namespace Vendor\Package;

class ClassName
{
```

```
    public $foo = null;
}
```

139.7.4 Method

- 所有方法都必须添加访问修饰符。
- 不该使用下划线作为前缀，来区分方法是 protected 或 private。
- 方法名称后一定不可有空格符，其开始花括号必须独占一行，结束花括号也必须在方法主体后单独成一行。参数左括号后和右括号前一定不可有空格。

一个标准的方法声明可参照以下范例，留意其括号、逗号、空格以及花括号的位置。

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

139.7.5 Parameter

参数列表中，每个逗号后面必须要有一个空格，而逗号前面一定不可有空格。

有默认值的参数，必须放到参数列表的末尾。

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function foo($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

参数列表可以分列成多行，这样，包括第一个参数在内的每个参数都必须单独成行。

拆分成多行的参数列表后，结束括号以及方法开始花括号必须写在同一行，中间用一个空格分隔。

```php
<?php
namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // 方法的内容
    }
}
```

139.7.6 Declare

需要添加 abstract 或 final 声明时，必须写在访问修饰符前，而 static 则必须写在其后。

```php
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

139.7.7 Method

方法及函数调用时，方法名或函数名与参数左括号之间一定不可有空格，参数右括号前也一定不可有空格。每个参数前一定不可有空格，但其后必须有一个空格。

```php
<?php
bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

参数可以分列成多行，此时包括第一个参数在内的每个参数都必须单独成行。

```
<?php
$foo->bar(
    $longArgument,
    $longerArgument,
    $muchLongerArgument
);
```

139.8 Control Structure

控制结构的基本规范如下：

- 控制结构关键词后必须有一个空格。
- 左括号 (后一定不可有空格。
- 右括号) 前也一定不可有空格。
- 右括号) 与开始花括号 { 间必须有一个空格。
- 结构体主体必须要有一次缩进。
- 结束花括号 } 必须在结构体主体后单独成行。

每个结构体的主体都必须被包含在成对的花括号之中，这能让结构体更加结构话，以及减少加入新行时，出错的可能性。

139.8.1 if/elseif/else

标准的 if 结构如下代码所示，请留意「括号」、「空格」以及「花括号」的位置，注意 else 和 elseif 都与前面的结束花括号在同一行。

```
<?php
if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

应该使用关键词 elseif 代替所有 else if ，以使得所有的控制关键字都像是单独的一个词。

139.8.2 switch/case

标准的 switch 结构如下代码所示，留意括号、空格以及花括号的位置。case 语句必须相对 switch 进行一次缩进，而 break 语句以及 case 内的其它语句都必须相对 case 进行一次缩

进。

如果存在非空的 case 直穿语句，主体里必须有类似 // no break 的注释。

```
<?php
switch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

139.8.3 while/do while

一个规范的 while 语句应该如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
while ($expr) {
    // structure body
}
```

标准的 do while 语句如下所示，同样的，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
do {
    // structure body;
} while ($expr);
```

139.8.4 for

标准的 for 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
```

```
for ($i = 0; $i < 10; $i++) {
    // for body
}
```

139.8.5 foreach

标准的 foreach 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
foreach ($iterable as $key => $value) {
    // foreach body
}
```

139.8.6 try/catch

标准的 try catch 语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
try {
    // try body
} catch (FirstExceptionType $e) {
    // catch body
} catch (OtherExceptionType $e) {
    // catch body
}
```

139.9 Closure

- 闭包声明时，关键词 function 后以及关键词 use 的前后都必须要有一个空格。
- 开始花括号必须写在声明的同一行，结束花括号必须紧跟主体结束的下一行。
- 参数列表和变量列表的左括号后以及右括号前，一定不可有空格。
- 参数和变量列表中，逗号前一定不可有空格，而逗号后必须要有空格。
- 闭包中有默认值的参数必须放到列表的后面。

标准的闭包声明语句如下所示，注意其「括号」、「空格」以及「花括号」的位置。

```
<?php
$closureWithArgs = function ($arg1, $arg2) {
    // body
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
```

```
    // body
};
```

参数列表以及变量列表可以分成多行，这样，包括第一个在内的每个参数或变量都必须单独成行，而列表的右括号与闭包的开始花括号必须放在同一行。

以下几个例子，包含了参数和变量列表被分成多行的多情况。

```
<?php
$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
    // body
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
    // body
```

```
};

$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
   // body
};
```

注意，闭包被直接用作函数或方法调用的参数时，以上规则仍然适用。

```
<?php
$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // body
    },
    $arg3
);
```

Chapter 140

PSR-3

本规范的主要目的,是为了让日志类库以简单通用的方式,通过接收一个 Psr\Log\LoggerInterface 对象来记录日志信息。

框架以及 CMS 内容管理系统如有需要，可以对此接口进行扩展，但需遵循本规范，这才能保证在使用第三方的类库文件时，日志接口仍能正常对接。

140.1 Overview

LoggerInterface 接口对外定义了八个方法，分别用来记录 RFC 5424 中定义的八个等级的日志：debug、info、notice、warning、error、critical、alert 以及 emergency 。

第九个方法——log，其第一个参数为记录的等级。可使用一个预先定义的等级常量作为参数来调用此方法，必须与直接调用以上八个方法具有相同的效果。如果传入的等级常量参数没有预先定义，则必须抛出 Psr\Log\InvalidArgumentException 类型的异常。在不确定的情况下，使用者不该使用未支持的等级常量来调用此方法。

140.1.1 Logging

以上每个方法都接受一个字符串类型或者是有 __toString() 方法的对象作为记录信息参数，这样，实现者就能把它当成字符串来处理，否则实现者必须自己把它转换成字符串。

记录信息参数可以携带占位符，实现者可以根据上下文将其它替换成相应的值。其中，占位符必须与上下文数组中的键名保持一致。

占位符的名称必须由一个左花括号 { 以及一个右括号 } 包含。但花括号与名称之间一定不可有空格符。

占位符的名称应该只由 A-Z、a-z、0-9、下划线 _、以及英文的句号. 组成，其它字符作为将来占位符规范的保留。

实现者可以通过对占位符采用不同的转义和转换策略，来生成最终的日志。而使用者在不知道上下文的前提下，不该提前转义占位符。

以下是一个占位符使用的例子：

```
/**
* 用上下文信息替换记录信息中的占位符
*/
function interpolate($message, array $context = array())
{
  // 构建一个花括号包含的键名的替换数组
  $replace = array();
  foreach ($context as $key => $val) {
      $replace['{' . $key . '}'] = $val;
  }

  // 替换记录信息中的占位符，最后返回修改后的记录信息。
  return strtr($message, $replace);
}

// 含有带花括号占位符的记录信息。
$message = "User {username} created";

// 带有替换信息的上下文数组，键名为占位符名称，键值为替换值。
$context = array('username' => 'bolivar');

// 输出 "Username bolivar created"
echo interpolate($message, $context);
```

140.1.2 Context

- 每个记录函数都接受一个上下文数组参数，用来装载字符串类型无法表示的信息。它可以装载任何信息，所以实现者必须确保能正确处理其装载的信息，对于其装载的数据，一定不可抛出异常，或产生 PHP 出错、警告或提醒信息（error、warning、notice）。
- 如需通过上下文参数传入了一个 Exception 对象，必须以 exception 作为键名。记录异常信息是很普遍的，所以如果它能够在记录类库的底层实现，就能够让实现者从异常信息中抽丝剥茧。当然，实现者在使用它时，必须确保键名为 exception 的键值是否真的是一个 Exception，毕竟它可以装载任何信息。

140.1.3 Helper

- Psr\Log\AbstractLogger 类使得只需继承它和实现其中的 log 方法，就能够很轻易地实现 LoggerInterface 接口，而另外八个方法就能够把记录信息和上下文信息传给它。
- 同样地，使用 Psr\Log\LoggerTrait 也只需实现其中的 log 方法。不过，需要特别注意的是，在 traits 可复用代码块还不能实现接口前，还需要 implement LoggerInterface。
- 在没有可用的日志记录器时，Psr\Log\NullLogger 接口可以为使用者提供一个备用的日志「黑洞」。不过，当上下文的构建非常消耗资源时，带条件检查的日志记录或许是更好的办法。
- Psr\Log\LoggerAwareInterface 接口仅包括一个 setLogger(LoggerInterface $logger) 方法，框架可以使用它实现自动连接任意的日志记录实例。
- Psr\Log\LoggerAwareTrait trait 可复用代码块可以在任何的类里面使用，只需通过它提供的 $this->logger，就可以轻松地实现等同的接口。
- Psr\Log\LogLevel 类装载了八个记录等级常量。

140.2 Package

上述的接口、类和相关的异常类，以及一系列的实现检测文件，都包含在 psr/log 文件包中。

140.3 Interface

140.3.1 Psr\Log\LoggerInterface

```
<?php

namespace Psr\Log;

/**
 * 日志记录实例
 *
 * 日志信息变量 —— message，**必须** 是一个字符串或是实现了 __toString() 方法的对象。
 *
 * 日志信息变量中 **可以** 包含格式如 “{foo}” （代表 foo）的占位符，
 * 它将会由上下文数组中键名为「foo」的键值替代。
 *
 * 上下文数组可以携带任意的数据，唯一的限制是，当它携带的是一个 exception 对象时，它的键名
    **必须** 是 "exception"。
 *
```

```
 * 详情可参阅： https://github.com/PizzaLiu/PHP-FIG/blob/master/PSR-3-logger-interface-cn
     .md
 */
interface LoggerInterface
{
    /**
     * 系统不可用
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function emergency($message, array $context = array());

    /**
     * **必须** 立刻采取行动
     *
     * 例如：在整个网站都垮掉了、数据库不可用了或者其他的情况下， **应该** 发送一条警报短信把
         你叫醒。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function alert($message, array $context = array());

    /**
     * 紧急情况
     *
     * 例如：程序组件不可用或者出现非预期的异常。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function critical($message, array $context = array());

    /**
     * 运行时出现的错误，不需要立刻采取行动，但必须记录下来以备检测。
     *
     * @param string $message
```

```
     * @param array $context
     * @return null
     */
    public function error($message, array $context = array());

    /**
     * 出现非错误性的异常。
     *
     * 例如：使用了被弃用的API、错误地使用了API或者非预想的不必要错误。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function warning($message, array $context = array());

    /**
     * 一般性重要的事件。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function notice($message, array $context = array());

    /**
     * 重要事件
     *
     * 例如：用户登录和SQL记录。
     *
     * @param string $message
     * @param array $context
     * @return null
     */
    public function info($message, array $context = array());

    /**
     * debug 详情
     *
     * @param string $message
     * @param array $context
```

```
     * @return null
     */
    public function debug($message, array $context = array());

    /**
     * 任意等级的日志记录
     *
     * @param mixed $level
     * @param string $message
     * @param array $context
     * @return null
     */
    public function log($level, $message, array $context = array());
}
```

140.3.2 Psr\Log\LoggerAwareInterface

```
<?php

namespace Psr\Log;

/**
 * logger-aware 定义实例
 */
interface LoggerAwareInterface
{
    /**
     * 设置一个日志记录实例
     *
     * @param LoggerInterface $logger
     * @return null
     */
    public function setLogger(LoggerInterface $logger);
}
```

140.3.3 Psr\Log\LogLevel

```
<?php

namespace Psr\Log;
```

```
/**
 * 日志等级常量定义
 */
class LogLevel
{
    const EMERGENCY = 'emergency';
    const ALERT   = 'alert';
    const CRITICAL = 'critical';
    const ERROR   = 'error';
    const WARNING = 'warning';
    const NOTICE  = 'notice';
    const INFO    = 'info';
    const DEBUG   = 'debug';
}
```

Chapter 141

PSR-4

141.1 Overview

PSR-4 是关于由文件路径自动载入对应类的相关规范，本规范是可互操作的，可以作为任一自动载入规范的补充，其中包括 PSR-0，此外，本 PSR 还包括自动载入的类对应的文件存放路径规范。

PSR-4 提供了一种命名空间的推荐使用方式，它提供了一个标准的文件、类和命名空间的使用惯例，可以让代码随插即用。

此处的「类」泛指所有的「Class 类」、「接口」、「traits 可复用代码块」以及其它类似结构。

一个完整的类名需具有以下结构:

```
\<命名空间>(\<子命名空间>)*\<类名>
```

- 完整的类名必须要有一个顶级命名空间，被称为”vendor namespace”;
- 完整的类名可以有一个或多个子命名空间;
- 完整的类名必须有一个最终的类名;
- 完整的类名中任意一部分中的下滑线都是没有特殊含义的;
- 完整的类名可以由任意大小写字母组成;
- 所有类名都必须是大小写敏感的。

当根据完整的类名载入相应的文件时:

- 完整的类名中，去掉最前面的命名空间分隔符，前面连续的一个或多个命名空间和子命名空间，作为「命名空间前缀」，其必须与至少一个「文件基目录」相对应;
- 紧接命名空间前缀后的子命名空间必须与相应的「文件基目录」相匹配，其中的命名空间分隔符将作为目录分隔符。
- 末尾的类名必须与对应的以.php 为后缀的文件同名。
- 自动加载器（autoloader）的实现一定不可抛出异常、一定不可触发任一级别的错误信

息以及不应该有返回值。

下表展示了符合规范完整类名、命名空间前缀和文件基目录所对应的文件路径。

表 141.1: PSR-4 示例

完整类名	命名空间前缀	文件基目录	文件路径
\Acme\Log\Writer\File_Writer	\Acme\Log\Writer	./acme-log-writer/lib/	./acme-log-writer/lib/File_Writer.php
\Aura\Web\Response\Status	\Aura\Web	/path/to/aura-web/src/	/path/to/aura-web/src/Response/Status.php
\Symfony\Core\Request	\Symfony\Core	/vendor/Symfony/Core/	/vendor/Symfony/Core/Request.php
\Zend\Acl	Zend	/usr/includes/Zend/	/usr/includes/Zend/Acl.php

141.2 Closure

```
<?php
/**
 * An example of a project-specific implementation.
 *
 * After registering this autoload function with SPL, the following line
 * would cause the function to attempt to load the \Foo\Bar\Baz\Qux class
 * from /path/to/project/src/Baz/Qux.php:
 *
 *     new \Foo\Bar\Baz\Qux;
 *
 * @param string $class The fully-qualified class name.
 * @return void
 */
spl_autoload_register(function ($class) {

    // project-specific namespace prefix
    $prefix = 'Foo\\Bar\\';

    // base directory for the namespace prefix
    $base_dir = __DIR__ . '/src/';

    // does the class use the namespace prefix?
```

```
    $len = strlen($prefix);
    if (strncmp($prefix, $class, $len) !== 0) {
        // no, move to the next registered autoloader
        return;
    }

    // get the relative class name
    $relative_class = substr($class, $len);

    // replace the namespace prefix with the base directory, replace namespace
    // separators with directory separators in the relative class name, append
    // with .php
    $file = $base_dir . str_replace('\\', '/', $relative_class) . '.php';

    // if the file exists, require it
    if (file_exists($file)) {
        require $file;
    }
});
```

141.3 Class

```
<?php
namespace Example;

/**
 * An example of a general-purpose implementation that includes the optional
 * functionality of allowing multiple base directories for a single namespace
 * prefix.
 *
 * Given a foo-bar package of classes in the file system at the following
 * paths ...
 *
 *    /path/to/packages/foo-bar/
 *        src/
 *            Baz.php           # Foo\Bar\Baz
 *            Qux/
 *                Quux.php      # Foo\Bar\Qux\Quux
 *        tests/
 *            BazTest.php       # Foo\Bar\BazTest
```

```
*           Qux/
*               QuuxTest.php  # Foo\Bar\Qux\QuuxTest
*
* ... add the path to the class files for the \Foo\Bar\ namespace prefix
* as follows:
*
*     <?php
*     // instantiate the loader
*     $loader = new \Example\Psr4AutoloaderClass;
*
*     // register the autoloader
*     $loader->register();
*
*     // register the base directories for the namespace prefix
*     $loader->addNamespace('Foo\Bar', '/path/to/packages/foo-bar/src');
*     $loader->addNamespace('Foo\Bar', '/path/to/packages/foo-bar/tests');
*
* The following line would cause the autoloader to attempt to load the
* \Foo\Bar\Qux\Quux class from /path/to/packages/foo-bar/src/Qux/Quux.php:
*
*     <?php
*     new \Foo\Bar\Qux\Quux;
*
* The following line would cause the autoloader to attempt to load the
* \Foo\Bar\Qux\QuuxTest class from /path/to/packages/foo-bar/tests/Qux/QuuxTest.php:
*
*     <?php
*     new \Foo\Bar\Qux\QuuxTest;
*/
class Psr4AutoloaderClass
{
    /**
     * An associative array where the key is a namespace prefix and the value
     * is an array of base directories for classes in that namespace.
     *
     * @var array
     */
    protected $prefixes = array();

    /**
     * Register loader with SPL autoloader stack.
```

```
     *
     * @return void
     */
    public function register()
    {
        spl_autoload_register(array($this, 'loadClass'));
    }

    /**
     * Adds a base directory for a namespace prefix.
     *
     * @param string $prefix The namespace prefix.
     * @param string $base_dir A base directory for class files in the
     * namespace.
     * @param bool $prepend If true, prepend the base directory to the stack
     * instead of appending it; this causes it to be searched first rather
     * than last.
     * @return void
     */
    public function addNamespace($prefix, $base_dir, $prepend = false)
    {
        // normalize namespace prefix
        $prefix = trim($prefix, '\\') . '\\';

        // normalize the base directory with a trailing separator
        $base_dir = rtrim($base_dir, DIRECTORY_SEPARATOR) . '/';

        // initialize the namespace prefix array
        if (isset($this->prefixes[$prefix]) === false) {
            $this->prefixes[$prefix] = array();
        }

        // retain the base directory for the namespace prefix
        if ($prepend) {
            array_unshift($this->prefixes[$prefix], $base_dir);
        } else {
            array_push($this->prefixes[$prefix], $base_dir);
        }
    }

    /**
```

```
     * Loads the class file for a given class name.
     *
     * @param string $class The fully-qualified class name.
     * @return mixed The mapped file name on success, or boolean false on
     * failure.
     */
    public function loadClass($class)
    {
        // the current namespace prefix
        $prefix = $class;

        // work backwards through the namespace names of the fully-qualified
        // class name to find a mapped file name
        while (false !== $pos = strrpos($prefix, '\\')) {

            // retain the trailing namespace separator in the prefix
            $prefix = substr($class, 0, $pos + 1);

            // the rest is the relative class name
            $relative_class = substr($class, $pos + 1);

            // try to load a mapped file for the prefix and relative class
            $mapped_file = $this->loadMappedFile($prefix, $relative_class);
            if ($mapped_file) {
                return $mapped_file;
            }

            // remove the trailing namespace separator for the next iteration
            // of strrpos()
            $prefix = rtrim($prefix, '\\');
        }

        // never found a mapped file
        return false;
    }

    /**
     * Load the mapped file for a namespace prefix and relative class.
     *
     * @param string $prefix The namespace prefix.
     * @param string $relative_class The relative class name.
```

```
     * @return mixed Boolean false if no mapped file can be loaded, or the
     * name of the mapped file that was loaded.
     */
    protected function loadMappedFile($prefix, $relative_class)
    {
        // are there any base directories for this namespace prefix?
        if (isset($this->prefixes[$prefix]) === false) {
            return false;
        }

        // look through base directories for this namespace prefix
        foreach ($this->prefixes[$prefix] as $base_dir) {

            // replace the namespace prefix with the base directory,
            // replace namespace separators with directory separators
            // in the relative class name, append with .php
            $file = $base_dir
                  . str_replace('\\', '/', $relative_class)
                  . '.php';

            // if the mapped file exists, require it
            if ($this->requireFile($file)) {
                // yes, we're done
                return $file;
            }
        }

        // never found it
        return false;
    }

    /**
     * If a file exists, require it from the file system.
     *
     * @param string $file The file to require.
     * @return bool True if the file exists, false if not.
     */
    protected function requireFile($file)
    {
        if (file_exists($file)) {
            require $file;
```

```
            return true;
        }
        return false;
    }
}
```

141.4 Unit Test

```
<?php
namespace Example\Tests;

class MockPsr4AutoloaderClass extends Psr4AutoloaderClass
{
    protected $files = array();

    public function setFiles(array $files)
    {
        $this->files = $files;
    }

    protected function requireFile($file)
    {
        return in_array($file, $this->files);
    }
}

class Psr4AutoloaderClassTest extends \PHPUnit_Framework_TestCase
{
    protected $loader;

    protected function setUp()
    {
        $this->loader = new MockPsr4AutoloaderClass;

        $this->loader->setFiles(array(
            '/vendor/foo.bar/src/ClassName.php',
            '/vendor/foo.bar/src/DoomClassName.php',
            '/vendor/foo.bar/tests/ClassNameTest.php',
            '/vendor/foo.bardoom/src/ClassName.php',
            '/vendor/foo.bar.baz.dib/src/ClassName.php',
```

```
            '/vendor/foo.bar.baz.dib.zim.gir/src/ClassName.php',
        ));

        $this->loader->addNamespace(
            'Foo\Bar',
            '/vendor/foo.bar/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar',
            '/vendor/foo.bar/tests'
        );

        $this->loader->addNamespace(
            'Foo\BarDoom',
            '/vendor/foo.bardoom/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar\Baz\Dib',
            '/vendor/foo.bar.baz.dib/src'
        );

        $this->loader->addNamespace(
            'Foo\Bar\Baz\Dib\Zim\Gir',
            '/vendor/foo.bar.baz.dib.zim.gir/src'
        );
    }

    public function testExistingFile()
    {
        $actual = $this->loader->loadClass('Foo\Bar\ClassName');
        $expect = '/vendor/foo.bar/src/ClassName.php';
        $this->assertSame($expect, $actual);

        $actual = $this->loader->loadClass('Foo\Bar\ClassNameTest');
        $expect = '/vendor/foo.bar/tests/ClassNameTest.php';
        $this->assertSame($expect, $actual);
    }

    public function testMissingFile()
```

```
    {
        $actual = $this->loader->loadClass('No_Vendor\No_Package\NoClass');
        $this->assertFalse($actual);
    }

    public function testDeepFile()
    {
        $actual = $this->loader->loadClass('Foo\Bar\Baz\Dib\Zim\Gir\ClassName');
        $expect = '/vendor/foo.bar.baz.dib.zim.gir/src/ClassName.php';
        $this->assertSame($expect, $actual);
    }

    public function testConfusion()
    {
        $actual = $this->loader->loadClass('Foo\Bar\DoomClassName');
        $expect = '/vendor/foo.bar/src/DoomClassName.php';
        $this->assertSame($expect, $actual);

        $actual = $this->loader->loadClass('Foo\BarDoom\ClassName');
        $expect = '/vendor/foo.bardoom/src/ClassName.php';
        $this->assertSame($expect, $actual);
    }
}
```

Chapter 142

PSR-6

PSR-6 的目标是创建一套通用的接口规范，能够让开发人员整合到现有框架和系统，而不需要去开发框架专属的适配器类。

142.1 Overview

缓存是提升应用性能的常用手段，为框架中最通用的功能，每个框架也都推出专属的、功能多样的缓存库。这些差别使得开发人员不得不学习多种系统，而很多可能是他们并不需要的功能。此外，缓存库的开发者同样面临着一个窘境，是只支持有限数量的几个框架还是创建一堆庞大的适配器类。

一个通用的缓存系统接口可以解决掉这些问题。库和框架的开发人员能够知道缓存系统会按照他们所预期的方式工作，缓存系统的开发人员只需要实现单一的接口，而不用去开发各种各样的适配器。

142.2 Definition

142.2.1 Calling Library

调用类库 (Calling Library) - 调用者，使用缓存服务的类库，这个类库调用缓存服务，调用的是此缓存接口规范的具体「实现类库」，调用者不需要知道任何「缓存服务」的具体实现。

142.2.2 Implementing Library

实现类库 (Implementing Library) - 此类库是对「缓存接口规范」的具体实现，封装起来的缓存服务，供「调用类库」使用。

实现类库必须提供 PHP 类来实现 Cache\CacheItemPoolInterface 和 Cache\CacheItemInterface 接口。实现类库必须支持最小的 TTL 功能，秒级别的精准度。

142.2.3 Expiration

过期时间 (Expiration) - 定义准确的过期时间点，一般为缓存存储发生的时间点加上 TTL 时间值，也可以指定一个 DateTime 对象。

假如一个缓存项的 TTL 设置为 300 秒, 保存于 1:30:00 , 那么缓存项的过期时间为 1:35:00。

实现类库可以让缓存项提前过期，但是必须在到达过期时间时立即把缓存项标示为过期。如果调用类库在保存一个缓存项的时候未设置「过期时间」、或者设置了 null 作为过期时间（或者 TTL 设置为 null），实现类库可以使用默认自行配置的一个时间。如果没有默认时间，实现类库必须把存储时间当做永久性存储，或者按照底层驱动能支持的最长时间作为保持时间。

142.2.4 TTL

生存时间值 (TTL - Time To Live) - 定义了缓存可以存活的时间，以秒为单位的整数值。

142.2.5 Key

长度大于 1 的字串，用作缓存项在缓存系统里的唯一标识符。

- 实现类库必须支持「键」规则 A-Z，a-z，0-9，_ 和. 任何顺序的 UTF-8 编码，长度小于 64 位。
- 实现类库可以支持更多的编码或者更长的长度，不过必须支持至少以上指定的编码和长度。
- 实现类库可自行实现对「键」的转义，但是必须保证能够无损的返回「键」字串。

以下的字串作为系统保留: {}()/\@:，一定不可作为「键」的命名支持。

142.2.6 Hit

命中 (Hit) - 一个缓存的命中，指的是当调用类库使用「键」在请求一个缓存项的时候，在缓存池里能找到对应的缓存项，并且此缓存项还未过期，并且此数据不会因为任何原因出现错误。

调用类库应该确保先验证下 isHit() 有命中后才调用 get() 获取数据。

142.2.7 Miss

未命中 (Miss) - 一个缓存未命中，是完全的上面描述的「命中」的相反。指的是当调用类库使用「键」在请求一个缓存项的时候，在缓存池里未能找到对应的缓存项，或者此缓存

项已经过期，或者此数据因为任何原因出现错误。

一个过期的缓存项，必须被当做未命中来对待。

142.2.8 Deferred

延迟 (Deferred) - 一个延迟的缓存，指的是这个缓存项可能不会立刻被存储到物理缓存池里。

一个缓存池对象可以对一个指定延迟的缓存项进行延迟存储，这样做的好处是可以利用一些缓存服务器提供的批量插入功能。缓存池必须能对所有延迟缓存最终能持久化，并且不会丢失。可以在调用类库还未发起保存请求之前就做持久化。

当调用类库调用 commit() 方法时，所有的延迟缓存都必须做持久化。实现类库可以自行决定使用什么逻辑来触发数据持久化，如对象的析构方法 (destructor) 内、调用 save() 时持久化、倒计时保存或者触及最大数量时保存等。

当请求一个延迟缓存项时，必须返回一个延迟，未持久化的缓存项对象。

142.3 Data Type

实现类库必须支持所有的可序列化的 PHP 数据类型，包含：

- 字符串 - 任何大小的 PHP 兼容字符串
- 整数 - PHP 支持的低于 64 位的有符号整数值
- 浮点数 - 所有的有符号浮点数
- 布尔 - true 和 false.
- Null - null 值
- 数组 - 各种形式的 PHP 数组
- 对象 (Object) - 所有的支持无损序列化和反序列化的对象，例如 `$o == unserialize(serialize($o))` 。

  对象可以使用 PHP 的 Serializable 接口，__sleep() 或者 __wakeup() 魔术方法，或者在合适的情况下，使用其他类似的语言特性。

所有存进实现类库的数据，都必须能做到原封不动的取出。连类型也必须是完全一致，如果存进缓存的是字符串 5，取出来的却是整数值 5 的话，可以算作严重的错误。实现类库可以使用 PHP 的「serialize()/unserialize() 方法」作为底层实现，不过不强迫这样做。对于它们的兼容性，以能支持所有数据类型作为基准线。

实在无法「完整取出」存入的数据的话，实现类库必须把「缓存丢失」标示作为返回，而不是损坏了的数据。

142.4 Concepts

142.4.1 Pool

缓存池包含缓存系统里所有缓存数据的集合。缓存池逻辑上是所有缓存项存储的仓库，所有存储进去的数据，都能从缓存池里取出来，所有的对缓存的操作，都发生在缓存池子里。

142.4.2 Items

一条缓存项在缓存池里代表了一对「键/值」对应的数据，「键」被视为每一个缓存项主键，是缓存项的唯一标识符，必须是不可变更的，当然，「值」可以任意变更。

142.5 Error Handling

缓存对应用性能起着至关重要的作用，但是，无论在任何情况下，缓存一定不可作为应用程序不可或缺的核心功能。

缓存系统里的错误一定不可导致应用程序故障，所以，实现类库一定不可抛出任何除了此接口规范定义的以外的异常，并且必须捕捉包括底层存储驱动抛出的异常，不让其冒泡至超出缓存系统内。

实现类库应该对此类错误进行记录，或者以任何形式通知管理员。

调用类库发起删除缓存项的请求，或者清空整个缓冲池子的请求，「键」不存在的话必须不能当成是有错误发生。后置条件是一样的，如果取数据时，「键」不存在的话必须不能当成是有错误发生。

142.6 Interface

142.6.1 CacheItemInterface

CacheItemInterface 定义了缓存系统里的一个缓存项。每一个缓存项必须有一个「键」与之相关联，此「键」通常是通过 Cache\CacheItemPoolInterface 来设置。

Cache\CacheItemInterface 对象把缓存项的存储进行了封装，每一个 Cache\CacheItemInterface 由一个 Cache\CacheItemPoolInterface 对象生成，CacheItemPoolInterface 负责一些必须的设置，并且给对象设置具有唯一性的「键」。

Cache\CacheItemInterface 对象必须能够存储和取出任何类型的定义有效的 PHP 数值。

调用类库一定不可擅自初始化「CacheItemInterface」对象，「缓存项」只能使用「CacheItemPoolInterface」对象的 getItem() 方法来获取。调用类库一定不可假设由一个实现类库创建的「缓存项」能被另一个实现类库完全兼容。

```php
namespace Psr\Cache;

/**
 * CacheItemInterface 定了缓存系统里对缓存项操作的接口
 */
interface CacheItemInterface
{
    /**
     * 返回当前缓存项的「键」
     *
     * 「键」由实现类库来加载，并且高层的调用者（如：CacheItemPoolInterface）
     * **应该** 能使用此方法来获取到「键」的信息。
     *
     * @return string
     *   当前缓存项的「键」
     */
    public function getKey();

    /**
     * 凭借此缓存项的「键」从缓存系统里面取出缓存项。
     *
     * 取出的数据 **必须** 跟使用 `set()` 存进去的数据是一模一样的。
     *
     * 如果 `isHit()` 返回 false 的话，此方法必须返回 `null`，需要注意的是 `null`
     * 本来就是一个合法的缓存数据，所以你 **应该** 使用 `isHit()` 方法来辨别到底是
     * "返回 null 数据" 还是 "缓存里没有此数据"。
     *
     * @return mixed
     *   此缓存项的「键」对应的「值」，如果找不到的话，返回 `null`
     */
    public function get();

    /**
     * 确认缓存项的检查是否命中。
     *
     * 注意：调用此方法和调用 `get()` 时 **一定不可** 有先后顺序之分。
     *
     * @return bool
     *   如果缓冲池里有命中的话，返回 `true`，反之返回 `false`
     */
    public function isHit();
```

```
/**
 * 为此缓存项设置「值」。
 *
 * 参数 $value 可以是所有能被 PHP 序列化的数据，序列化的逻辑
 * 需要在实现类库里书写。
 *
 * @param mixed $value
 *   将被存储的可序列化的数据。
 *
 * @return static
 *   返回当前对象。
 */
public function set($value);

/**
 * 设置缓存项的准确过期时间点。
 *
 * @param \DateTimeInterface $expiration
 *
 *   过期的准确时间点，过了这个时间点后，缓存项就 **必须** 被认为是过期了的。
 *   如果明确的传参 `null` 的话，**可以** 使用一个默认的时间。
 *   如果没有设置的话，缓存 **应该** 存储到底层实现的最大允许时间。
 *
 * @return static
 *   返回当前对象。
 */
public function expiresAt($expiration);

/**
 * 设置缓存项的过期时间。
 *
 * @param int|\DateInterval $time
 *   以秒为单位的过期时长，过了这段时间后，缓存项就 **必须** 被认为是过期了的。
 *   如果明确的传参 `null` 的话，**可以** 使用一个默认的时间。
 *   如果没有设置的话，缓存 **应该** 存储到底层实现的最大允许时间。
 *
 * @return static
 *   返回当前对象
 */
public function expiresAfter($time);
```

```
}
```

142.6.2 CacheItemPoolInterface

Cache\CacheItemPoolInterface 的主要目的是从调用类库接收「键」，然后返回对应的 Cache\CacheItemInterface 对象。

此接口也是作为主要的，与整个缓存集合交互的方式。所有的配置和初始化由实现类库自行实现。

```
namespace Psr\Cache;

/**
 * CacheItemPoolInterface 生成 CacheItemInterface 对象
 */
interface CacheItemPoolInterface
{
    /**
     * 返回「键」对应的一个缓存项。
     *
     * 此方法 **必须** 返回一个 CacheItemInterface 对象，即使是找不到对应的缓存项
     * 也 **一定不可** 返回 `null`。
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
     * @throws InvalidArgumentException
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会被抛出。
     *
     * @return CacheItemInterface
     *   对应的缓存项。
     */
    public function getItem($key);

    /**
     * 返回一个可供遍历的缓存项集合。
     *
     * @param array $keys
     *   由一个或者多个「键」组成的数组。
     *
     * @throws InvalidArgumentException
```

```
     *   如果 $keys 里面有哪个「键」不是合法，\Psr\Cache\InvalidArgumentException 异常
     *   会被抛出。
     *
     * @return array|\Traversable
     *   返回一个可供遍历的缓存项集合，集合里每个元素的标识符由「键」组成，即使即使是找不到对
     *   的缓存项，也要返回一个「CacheItemInterface」对象到对应的「键」中。
     *   如果传参的数组为空，也需要返回一个空的可遍历的集合。
     */
    public function getItems(array $keys = array());

    /**
     * 检查缓存系统中是否有「键」对应的缓存项。
     *
     * 注意：此方法应该调用 `CacheItemInterface::isHit()` 来做检查操作，而不是
     * `CacheItemInterface::get()`
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
     * @throws InvalidArgumentException
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会被抛出。
     *
     * @return bool
     *   如果存在「键」对应的缓存项即返回 true，否则 false
     */
    public function hasItem($key);

    /**
     * 清空缓冲池
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function clear();

    /**
     * 从缓冲池里移除某个缓存项
     *
     * @param string $key
     *   用来搜索缓存项的「键」。
     *
```

```
     * @throws InvalidArgumentException
     *   如果 $key 不是合法的值，\Psr\Cache\InvalidArgumentException 异常会被抛出。
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function deleteItem($key);

    /**
     * 从缓冲池里移除多个缓存项
     *
     * @param array $keys
     *   由一个或者多个「键」组成的数组。
     *
     * @throws InvalidArgumentException
     *   如果 $keys 里面有哪个「键」不是合法，\Psr\Cache\InvalidArgumentException 异常
     *   会被抛出。
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function deleteItems(array $keys);

    /**
     * 立刻为「CacheItemInterface」对象做数据持久化。
     *
     * @param CacheItemInterface $item
     *   将要被存储的缓存项
     *
     * @return bool
     *   成功返回 true，有错误发生返回 false
     */
    public function save(CacheItemInterface $item);

    /**
     * 稍后为「CacheItemInterface」对象做数据持久化。
     *
     * @param CacheItemInterface $item
     *   将要被存储的缓存项
     *
     * @return bool
```

```
     *  成功返回 true，有错误发生返回 false
     */
    public function saveDeferred(CacheItemInterface $item);

    /**
     * 提交所有的正在队列里等待的请求到数据持久层，配合 `saveDeferred()` 使用
     *
     * @return bool
     *  成功返回 true，有错误发生返回 false
     */
    public function commit();
}
```

142.6.3 CacheException

此异常用于缓存系统发生的所有严重错误，包括但不限制于缓存系统配置，如连接到缓存服务器出错、错误的用户身份认证等。

所有的实现类库抛出的异常都必须实现此接口。

```
namespace Psr\Cache;

/**
 * 被所有的实现类库抛出的异常继承的「异常接口」
 */
interface CacheException
{
}
```

142.6.4 InvalidArgumentException

```
namespace Psr\Cache;

/**
 * 传参错误抛出的异常接口
 *
 * 当一个错误或者非法的传参发生时，**必须** 抛出一个继承了
 * Psr\Cache\InvalidArgumentException 的异常
 */
interface InvalidArgumentException extends CacheException
{
}
```

Chapter 143

PSR-7

PSR-7 描述了 RFC 7230 和 RFC 7231 HTTP 消息传递的接口，还有 RFC 3986 里对 HTTP 消息的 URIs 使用。

143.1 Overview

143.1.1 HTTP Message

HTTP 消息是 Web 技术发展的基础。浏览器或 HTTP 客户端如 curl 生成发送 HTTP 请求消息到 Web 服务器，Web 服务器响应 HTTP 请求。服务端的代码接受 HTTP 请求消息后返回 HTTP 响应消息。

通常 HTTP 消息对于终端用户来说是不可见的，但是作为 Web 开发者，我们需要知道 HTTP 机制，如何发起、构建、取用还有操纵 HTTP 消息，知道这些原理，以助我们刚好的完成开发任务，无论这个任务是发起一个 HTTP 请求，或者处理传入的请求。

143.1.2 HTTP Request

每一个 HTTP 请求都有专属的格式：

```
POST /path HTTP/1.1
Host: example.com

foo=bar&baz=bat
```

按照顺序，第一行的各个字段意义为：HTTP 请求方法、请求的目标地址（通常是一个绝对路径的 URI 或者路径），HTTP 协议。

接下来是 HTTP 头信息，在这个例子中：目的主机。接下来是空行，然后是消息内容。

143.1.3 HTTP Response

HTTP 返回消息有类似的结构:

```
HTTP/1.1 200 OK
Content-Type: text/plain

这是返回的消息内容
```

按照顺序，第一行为状态行，包括 HTTP 协议版本，HTTP 状态码，描述文本。

和 HTTP 请求类似的，接下来是 HTTP 头信息，在这个例子中：内容类型。接下来是空行，然后是消息内容。

PSR-7 文档探讨的是 HTTP 请求消息接口，和构建 HTTP 消息需要的元素数据定义。

143.2 Specification

143.2.1 Message

一个 HTTP 消息，指定是一个从客户端发往服务器端的请求，或者是服务器端返回客户端的响应。对应的两个消息接口：Psr\Http\Message\RequestInterface 和 Psr\Http\Message\ResponseInterface。

这个两个接口都继承于 Psr\Http\Message\MessageInterface。虽然你可以实现 Psr\Http\Message\MessageInte 接口，但是，最重要的，你必须实现 Psr\Http\Message\RequestInterface 和 Psr\Http\Message\ResponseInterface 接口。

从现在开始，为了行文的方便，我们提到这些接口的时候，都会把命名空间 Psr\Http\Message 去除掉。

143.2.2 Header

大小写不敏感的字段名字

HTTP 消息包含大小写不敏感头信息。使用 MessageInterface 接口来设置和获取头信息，大小写不敏感的定义在于，如果你设置了一个 Foo 的头信息，foo 的值会被重写，你也可以通过 foo 来拿到 FoO 头对应的值。

```php
$message = $message->withHeader('foo', 'bar');

echo $message->getHeaderLine('foo');
// 输出: bar

echo $message->getHeaderLine('FOO');
// 输出: bar
```

```
$message = $message->withHeader('fOO', 'baz');
echo $message->getHeaderLine('foo');
// 输出：baz
```

虽然头信息可以用大小写不敏感的方式取出，但是接口实现类必须保持自己的大小写规范，特别是用 getHeaders() 方法输出的内容。

因为一些非标准的 HTTP 应用程序，可能会依赖于大小写敏感的头信息，所有在此我们把主宰 HTTP 大小写的权利开放出来，以适用不同的场景。

为了适用一个 HTTP 「键」可以对应多条数据的情况，我们使用字符串配合数组来实现，你可以从一个 MessageInterface 取出数组或字符串，使用 getHeaderLine($name) 方法可以获取通过逗号分割的不区分大小写的字符串形式的所有值。也可以通过 getHeader($name) 获取数组形式头信息的所有值。

```
$message = $message
    ->withHeader('foo', 'bar')
    ->withAddedHeader('foo', 'baz');

$header = $message->getHeaderLine('foo');
// $header 包含: 'bar, baz'

$header = $message->getHeader('foo');
// ['bar', 'baz']
```

注意，并不是所有的头信息都可以适用逗号分割（例如 Set-Cookie），当处理这种头信息时候，MessageInterace 的继承类应该使用 getHeader($name) 方法来获取这种多值的情况。

在请求中，Host 头信息通常和 URI 的 host 信息，还有建立起 TCP 连接使用的 Host 信息一致。然而，HTTP 标准规范允许主机 host 信息与其他两个不一样。

在构建请求的时候，如果 host 头信息未提供的话，实现类库必须尝试着从 URI 中提取 host 信息。

RequestInterface::withUri() 会默认的，从传参的 UriInterface 实例中提取 host ，并替代请求中原有的 host 信息。

你可以提供传参第二个参数为 true 来保证返回的消息实例中，原有的 host 头信息不会被替代掉。

以下表格说明了当 withUri() 的第二个参数被设置为 true 的时，返回的消息实例中调用 getHeaderLine('Host') 方法会返回的内容：

表 143.1: Host 信息示例

请求 Host 头信息	请求 URI 中的 Host 信息	传参进去 URI 的 Host	结果
''	''	''	''
''	foo.com	''	foo.com
''	foo.com	bar.com	foo.com
foo.com	''	bar.com	foo.com
foo.com	bar.com	baz.com	foo.com

- 请求 Host 头信息 - 当前请求的 Host 头信息
- 请求 URI 中的 Host 信息 - 当前请求 URI 中的 Host 信息
- 传参进去 URI 的 Host - 通过 withUri() 传参进入的 URI 中的 host 信息

143.2.3 Stream

HTTP 消息包含开始的一行、头信息、还有消息的内容。HTTP 的消息内容有时候可以很小，有时候确是非常巨大。尝试使用字符串的形式来展示消息内容，会消耗大量的内存，使用数据流的形式来读取消息可以解决此问题。

StreamInterface 接口用来隐藏具体的数据流读写实现。在一些情况下，消息类型的读取方式为字符串是能容许的，可以使用 php://memory 或者 php://temp。

StreamInterface 暴露出来几个接口，这些接口允许你读取、写入，还有高效的遍历内容。

数据流使用这个三个接口来阐明对他们的操作能力：isReadable()、isWritable() 和 isSeekable()。这些方法可以让数据流的操作者得知数据流能否能提供他们想要的功能。

每一个数据流的实例，都会有多种功能：可以只读、可以只写、可以读和写，可以随机读取，可以按顺序读取等。

最终，StreamInterface 定义了一个 __toString() 的方法，用来一次性以字符串的形式输出所有消息内容。

与请求和响应的接口不同的是，StreamInterface 并不强调不可修改性。因为在 PHP 的实现内，基本上没有办法保证不可修改性，因为指针的指向，内容的变更等状态，都是不可控的。

作为读取者，可以调用只读的方法来返回数据流，以最大程度上保证数据流的不可修改性。使用者要时刻明确的知道数据流的可修改性，建议把数据流附加到消息实例中，来强迫不可修改的特性。

143.3 Request Target

Per RFC 7230, request messages contain a ”request-target” as the second segment of the request line. The request target can be one of the following forms:

- origin-form, which consists of the path, and, if present, the query string; this is often referred to as a relative URL. Messages as transmitted over TCP typically are of origin-form; scheme and authority data are usually only present via CGI variables.
- absolute-form, which consists of the scheme, authority (”[user-info@]host[:port]”, where items in brackets are optional), path (if present), query string (if present), and fragment (if present). This is often referred to as an absolute URI, and is the only form to specify a URI as detailed in RFC 3986. This form is commonly used when making requests to HTTP proxies.
- authority-form, which consists of the authority only. This is typically used in CONNECT requests only, to establish a connection between an HTTP client and a proxy server.
- asterisk-form, which consists solely of the string *, and which is used with the OPTIONS method to determine the general capabilities of a web server.

Aside from these request-targets, there is often an ’effective URL’ which is separate from the request target. The effective URL is not transmitted within an HTTP message, but it is used to determine the protocol (http/https), port and hostname for making the request.

The effective URL is represented by UriInterface. UriInterface models HTTP and HTTPS URIs as specified in RFC 3986 (the primary use case). The interface provides methods for interacting with the various URI parts, which will obviate the need for repeated parsing of the URI. It also specifies a __toString() method for casting the modeled URI to its string representation.

When retrieving the request-target with getRequestTarget(), by default this method will use the URI object and extract all the necessary components to construct the origin-form. The origin-form is by far the most common request-target.

If it’s desired by an end-user to use one of the other three forms, or if the user wants to explicitly override the request-target, it is possible to do so with withRequestTarget().

Calling this method does not affect the URI, as it is returned from getUri().

For example, a user may want to make an asterisk-form request to a server:

```
$request = $request
   ->withMethod('OPTIONS')
   ->withRequestTarget('*')
   ->withUri(new Uri('https://example.org/'));
```

This example may ultimately result in an HTTP request that looks like this:

```
OPTIONS * HTTP/1.1
```

But the HTTP client will be able to use the effective URL (from getUri()), to determine the protocol, hostname and TCP port.

An HTTP client MUST ignore the values of Uri::getPath() and Uri::getQuery(), and instead use the value returned by getRequestTarget(), which defaults to concatenating these two values.

Clients that choose to not implement 1 or more of the 4 request-target forms, MUST still use getRequestTarget(). These clients MUST reject request-targets they do not support, and MUST NOT fall back on the values from getUri().

RequestInterface provides methods for retrieving the request-target or creating a new instance with the provided request-target. By default, if no request-target is specifically composed in the instance, getRequestTarget() will return the origin-form of the composed URI (or "/" if no URI is composed). withRequestTarget($requestTarget) creates a new instance with the specified request target, and thus allows developers to create request messages that represent the other three request-target forms (absolute-form, authority-form, and asterisk-form). When used, the composed URI instance can still be of use, particularly in clients, where it may be used to create the connection to the server.

143.4 Server-side Requests

RequestInterface provides the general representation of an HTTP request message. However, server-side requests need additional treatment, due to the nature of the server-side environment. Server-side processing needs to take into account Common Gateway Interface (CGI), and, more specifically, PHP's abstraction and extension of CGI via its Server APIs (SAPI). PHP has provided simplification around input marshaling via superglobals such as:

- $_COOKIE, which deserializes and provides simplified access for HTTP cookies.
- $_GET, which deserializes and provides simplified access for query string arguments.
- $_POST, which deserializes and provides simplified access for urlencoded parameters submitted via HTTP POST; generically, it can be considered the results of parsing the message body.
- $_FILES, which provides serialized metadata around file uploads.
- $_SERVER, which provides access to CGI/SAPI environment variables, which commonly include the request method, the request scheme, the request URI, and headers.

ServerRequestInterface extends RequestInterface to provide an abstraction around these various superglobals. This practice helps reduce coupling to the superglobals by consumers, and encourages and promotes the ability to test request consumers.

The server request provides one additional property, "attributes", to allow consumers the ability to introspect, decompose, and match the request against application-specific rules (such as path matching, scheme matching, host matching, etc.). As such, the server request can also provide messaging between multiple request consumers.

143.5 Upload Files

ServerRequestInterface specifies a method for retrieving a tree of upload files in a normalized structure, with each leaf an instance of UploadedFileInterface.

The $_FILES superglobal has some well-known problems when dealing with arrays of file inputs. As an example, if you have a form that submits an array of files —e.g., the input name "files", submitting files[0] and files[1] —PHP will represent this as:

```
array(
    'files' => array(
        'name' => array(
            0 => 'file0.txt',
            1 => 'file1.html',
        ),
        'type' => array(
            0 => 'text/plain',
            1 => 'text/html',
        ),
        /* etc. */
    ),
)
```

instead of the expected:

```
array(
    'files' => array(
        0 => array(
            'name' => 'file0.txt',
            'type' => 'text/plain',
            /* etc. */
        ),
        1 => array(
            'name' => 'file1.html',
            'type' => 'text/html',
            /* etc. */
        ),
    ),
)
```

The result is that consumers need to know this language implementation detail, and write code for gathering the data for a given upload.

Additionally, scenarios exist where $_FILES is not populated when file uploads occur:

- When the HTTP method is not POST.

- When unit testing.
- When operating under a non-SAPI environment, such as ReactPHP.

In such cases, the data will need to be seeded differently. As examples:

- A process might parse the message body to discover the file uploads. In such cases, the implementation may choose not to write the file uploads to the file system, but instead wrap them in a stream in order to reduce memory, I/O, and storage overhead.
- In unit testing scenarios, developers need to be able to stub and/or mock the file upload metadata in order to validate and verify different scenarios.

getUploadedFiles() provides the normalized structure for consumers. Implementations are expected to:

- Aggregate all information for a given file upload, and use it to populate a Psr\Http\Message\UploadedFileInterfac instance.
- Re-create the submitted tree structure, with each leaf being the appropriate Psr\Http\Message\UploadedFileInter instance for the given location in the tree.

The tree structure referenced should mimic the naming structure in which files were submitted.

In the simplest example, this might be a single named form element submitted as:

```
<input type="file" name="avatar" />
```

In this case, the structure in $_FILES would look like:

```
array(
    'avatar' => array(
        'tmp_name' => 'phpUxcOty',
        'name' => 'my-avatar.png',
        'size' => 90996,
        'type' => 'image/png',
        'error' => 0,
    ),
)
```

The normalized form returned by getUploadedFiles() would be:

```
array(
    'avatar' => /* UploadedFileInterface instance */
)
```

In the case of an input using array notation for the name:

```
<input type="file" name="my-form[details][avatar]" />
```

$_FILES ends up looking like this:

```
array(
    'my-form' => array(
        'details' => array(
            'avatar' => array(
                'tmp_name' => 'phpUxcOty',
                'name' => 'my-avatar.png',
                'size' => 90996,
                'type' => 'image/png',
                'error' => 0,
            ),
        ),
    ),
)
```

And the corresponding tree returned by getUploadedFiles() should be:

```
array(
    'my-form' => array(
        'details' => array(
            'avatar' => /* UploadedFileInterface instance */
        ),
    ),
)
```

In some cases, you may specify an array of files:

```
Upload an avatar: <input type="file" name="my-form[details][avatars][]" />
Upload an avatar: <input type="file" name="my-form[details][avatars][]" />
```

(As an example, JavaScript controls might spawn additional file upload inputs to allow uploading multiple files at once.)

In such a case, the specification implementation must aggregate all information related to the file at the given index. The reason is because $_FILES deviates from its normal structure in such cases:

```
array(
    'my-form' => array(
        'details' => array(
            'avatars' => array(
                'tmp_name' => array(
                    0 => '...',
                    1 => '...',
                    2 => '...',
                ),
                'name' => array(
```

```
                0 => '...',
                1 => '...',
                2 => '...',
            ),
            'size' => array(
                0 => '...',
                1 => '...',
                2 => '...',
            ),
            'type' => array(
                0 => '...',
                1 => '...',
                2 => '...',
            ),
            'error' => array(
                0 => '...',
                1 => '...',
                2 => '...',
            ),
        ),
    ),
),
```

The above $_FILES array would correspond to the following structure as returned by getUploadedFiles():

```
array(
    'my-form' => array(
        'details' => array(
            'avatars' => array(
                0 => /* UploadedFileInterface instance */,
                1 => /* UploadedFileInterface instance */,
                2 => /* UploadedFileInterface instance */,
            ),
        ),
    ),
)
```

Consumers would access index 1 of the nested array using:

```
$request->getUploadedFiles()['my-form']['details']['avatars'][1];
```

Because the uploaded files data is derivative (derived from $_FILES or the request body), a mutator method, withUploadedFiles(), is also present in the interface, allowing delegation of the normalization to another process.

In the case of the original examples, consumption resembles the following:

```
$file0 = $request->getUploadedFiles()['files'][0];
$file1 = $request->getUploadedFiles()['files'][1];

printf(
    "Received the files %s and %s",
    $file0->getClientFilename(),
    $file1->getClientFilename()
);

// "Received the files file0.txt and file1.html"
```

This proposal also recognizes that implementations may operate in non-SAPI environments. As such, UploadedFileInterface provides methods for ensuring operations will work regardless of environment. In particular:

- moveTo($targetPath) is provided as a safe and recommended alternative to calling move_uploaded_file() directly on the temporary upload file. Implementations will detect the correct operation to use based on environment.
- getStream() will return a StreamInterface instance. In non-SAPI environments, one proposed possibility is to parse individual upload files into php://temp streams instead of directly to files; in such cases, no upload file is present. getStream() is therefore guaranteed to work regardless of environment.

As examples:

```
// Move a file to an upload directory
$filename = sprintf(
    '%s.%s',
    create_uuid(),
    pathinfo($file0->getClientFilename(), PATHINFO_EXTENSION)
);
$file0->moveTo(DATA_DIR . '/' . $filename);

// Stream a file to Amazon S3.
// Assume $s3wrapper is a PHP stream that will write to S3, and that
// Psr7StreamWrapper is a class that will decorate a StreamInterface as a PHP
// StreamWrapper.
$stream = new Psr7StreamWrapper($file1->getStream());
```

```
stream_copy_to_stream($stream, $s3wrapper);
```

143.6 Package

上面讨论的接口和类库已经整合成为扩展包：psr/http-message。

143.7 Interface

143.7.1 Psr\Http\MessageInterface

```
<?php
namespace Psr\Http\Message;

/**
 *
 * HTTP 消息值得是客户端发起的「请求」和服务器端返回的「响应」，此接口
 * 定义了他们通用的方法。
 *
 * HTTP 消息是被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套
 * 机制，在内部保持好原有的内容，然后把修改状态后的信息返回。
 *
 * @see http://www.ietf.org/rfc/rfc7230.txt
 * @see http://www.ietf.org/rfc/rfc7231.txt
 */
interface MessageInterface
{
    /**
     * 获取字符串形式的 HTTP 协议版本信息
     *
     * 字符串必须包含 HTTP 版本数字，如："1.1", "1.0"。
     *
     * @return string HTTP 协议版本
     */
    public function getProtocolVersion();

    /**
     * 返回指定 HTTP 版本号的消息实例。
     *
     * 传参的版本号必须 **只** 包含 HTTP 版本数字，如："1.1", "1.0"。
     *
```

```
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息对象，然后返回
     * 一个新的带有传参进去的 HTTP 版本的实例
     *
     * @param string $version HTTP 版本信息
     * @return self
     */
    public function withProtocolVersion($version);

    /**
     * 获取所有的头信息
     *
     * 返回的二维数组中，第一维数组的「键」代表单条头信息的名字，「值」是
     * 以数据形式返回的，见以下实例：
     *
     *    // 把「值」的数据当成字串打印出来
     *    foreach ($message->getHeaders() as $name => $values) {
     *        echo $name . ': ' . implode(', ', $values);
     *    }
     *
     *    // 迭代的循环二维数组
     *    foreach ($message->getHeaders() as $name => $values) {
     *        foreach ($values as $value) {
     *            header(sprintf('%s: %s', $name, $value), false);
     *        }
     *    }
     *
     * 虽然头信息是没有大小写之分，但是使用 `getHeaders()` 会返回保留了原本
     * 大小写形式的内容。
     *
     * @return string[][] 返回一个两维数组，第一维数组的「键」 **必须** 为单条头信息的
     *    名称，对应的是由字串组成的数组，请注意，对应的「值」 **必须** 是数组形式的。
     */
    public function getHeaders();

    /**
     * 检查是否头信息中包含有此名称的值，不区分大小写
     *
     * @param string $name 不区分大小写的头信息名称
     * @return bool 找到返回 true，未找到返回 false
     */
    public function hasHeader($name);
```

```
/**
 * 根据给定的名称，获取一条头信息，不区分大小写，以数组形式返回
 *
 * 此方法以数组形式返回对应名称的头信息。
 *
 * 如果没有对应的头信息，**必须** 返回一个空数组。
 *
 * @param string $name 不区分大小写的头部字段名称。
 * @return string[] 返回头信息中，对应名称的，由字符串组成的数组值，如果没有对应
 *  的内容，**必须** 返回空数组。
 */
public function getHeader($name);

/**
 * 根据给定的名称，获取一条头信息，不区分大小写，以逗号分隔的形式返回
 *
 * 此方法返回所有对应的头信息，并将其使用逗号分隔的方法拼接起来。
 *
 * 注意：不是所有的头信息都可使用逗号分隔的方法来拼接，对于那些头信息，请使用
 * `getHeader()` 方法来获取。
 *
 * 如果没有对应的头信息，此方法 **必须** 返回一个空字符串。
 *
 * @param string $name 不区分大小写的头部字段名称。
 * @return string 返回头信息中，对应名称的，由逗号分隔组成的字串，如果没有对应
 *  的内容，**必须** 返回空字符串。
 */
public function getHeaderLine($name);

/**
 * 返回指定头信息「键/值」对的消息实例。
 *
 * 虽然头信息是不区分大小写的，但是此方法必须保留其传参时的大小写状态，并能够在
 * 调用 `getHeaders()` 的时候被取出。
 *
 * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息对象，然后返回
 * 一个新的带有传参进去头信息的实例
 *
 * @param string $name Case-insensitive header field name.
 * @param string|string[] $value Header value(s).
```

```
     * @return self
     * @throws \InvalidArgumentException for invalid header names or values.
     */
    public function withHeader($name, $value);

    /**
     * 返回一个头信息增量的 HTTP 消息实例。
     *
     * 原有的头信息会被保留，新的值会作为增量加上，如果头信息不存在的话，会被加上。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息对象，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param string $name 不区分大小写的头部字段名称。
     * @param string|string[] $value 头信息对应的值。
     * @return self
     * @throws \InvalidArgumentException 头信息字段名称非法时会被抛出。
     * @throws \InvalidArgumentException 头信息的值非法的时候，会被抛出。
     */
    public function withAddedHeader($name, $value);

    /**
     * 返回被移除掉指定头信息的 HTTP 消息实例。
     *
     * 头信息字段在解析的时候，**必须** 保证是不区分大小写的。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息对象，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param string $name 不区分大小写的头部字段名称。
     * @return self
     */
    public function withoutHeader($name);

    /**
     * 获取 HTTP 消息的内容。
     *
     * @return StreamInterface 以数据流的形式返回。
     */
    public function getBody();
```

```
    /**
     * 返回拼接了内容的 HTTP 消息实例。
     *
     * 内容 **必须** 是 StreamInterface 接口的实例。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息对象，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param StreamInterface $body 数据流形式的内容。
     * @return self
     * @throws \InvalidArgumentException 当消息内容不正确的时候。
     */
    public function withBody(StreamInterface $body);
}
```

143.7.2 Psr\Http\Message\RequestInterface

```
<?php
namespace Psr\Http\Message;

/**
 * 代表客户端请求的 HTTP 消息对象。
 *
 * 根据规范，每一个 HTTP 请求都包含以下信息：
 *
 * - HTTP 协议版本号 (Protocol version)
 * - HTTP 请求方法 (HTTP method)
 * - URI
 * - 头信息 (Headers)
 * - 消息内容 (Message body)
 *
 * 在构造 HTTP 请求对象的时候，实现类库 **必须** 从给出的 URI 中去提取 HOST 信息。
 *
 * HTTP 请求是被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套机制，在内部保
 * 持好原有的内容，然后把修改状态后的，新的 HTTP 请求实例返回。
 */
interface RequestInterface extends MessageInterface
{
    /**
     * 获取消息请求的目标。
     *
```

```
     * 在大部分情况下，此方法会返回完整的 URI，除非 `withRequestTarget()` 被设置过。
     *
     * 如果没有提供 URI，并且没有提供任何的请求目标，此方法 **必须** 返回 "/"。
     *
     * @return string
     */
    public function getRequestTarget();

    /**
     * 返回一个指定目标的请求实例。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 请求实例，然后返回
     * 一个新的修改过的 HTTP 请求实例。
     *
     * @see 关于请求目标的各种允许的格式，请见 http://tools.ietf.org/html/rfc7230#section
         -2.7
     *
     * @param mixed $requestTarget
     * @return self
     */
    public function withRequestTarget($requestTarget);

    /**
     * 获取当前请求使用的 HTTP 方法
     *
     * @return string HTTP 方法字符串
     */
    public function getMethod();

    /**
     * 返回更改了请求方法的消息实例。
     *
     * 虽然，在大部分情况下，HTTP 请求方法都是使用大写字母来标示的，但是，实现类库 **一定不
         可**
     * 修改用户传参的大小格式。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 请求实例，然后返回
     * 一个新的修改过的 HTTP 请求实例。
     *
     * @param string $method 大小写敏感的方法名
     * @return self
```

```
     * @throws \InvalidArgumentException 当非法的 HTTP 方法名传入时会抛出异常。
     */
    public function withMethod($method);

    /**
     * 获取 URI 实例。
     *
     * 此方法必须返回 `UriInterface` 的 URI 实例。
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.3
     * @return UriInterface 返回与当前请求相关 `UriInterface` 类型的 URI 实例。
     */
    public function getUri();

    /**
     * 返回修改了 URI 的消息实例。
     *
     * 当传入的 `URI` 包含有 `HOST` 信息时，**必须** 更新 `HOST` 头信息，如果 `URI`
     * 实例没有附带 `HOST` 信息，任何之前存在的 `HOST` 信息 **必须** 作为候补，应用
     * 更改到返回的消息实例里。
     *
     * 你可以通过传入第二个参数来，来干预方法的处理，当 `$preserveHost` 设置为 `true`
     * 的时候，会保留原来的 `HOST` 信息。
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 请求实例，然后返回
     * 一个新的修改过的 HTTP 请求实例。
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.3
     * @param UriInterface $uri `UriInterface` 类型的 URI 实例
     * @param bool $preserveHost 是否保留原有的 HOST 头信息
     * @return self
     */
    public function withUri(UriInterface $uri, $preserveHost = false);
}
```

143.7.3 Psr\Http\Message\ServerRequestInterface

```
<?php
namespace Psr\Http\Message;

/**
```

```
 * Representation of an incoming, server-side HTTP request.
 *
 * Per the HTTP specification, this interface includes properties for
 * each of the following:
 *
 * - Protocol version
 * - HTTP method
 * - URI
 * - Headers
 * - Message body
 *
 * Additionally, it encapsulates all data as it has arrived to the
 * application from the CGI and/or PHP environment, including:
 *
 * - The values represented in $_SERVER.
 * - Any cookies provided (generally via $_COOKIE)
 * - Query string arguments (generally via $_GET, or as parsed via parse_str())
 * - Upload files, if any (as represented by $_FILES)
 * - Deserialized body parameters (generally from $_POST)
 *
 * $_SERVER values MUST be treated as immutable, as they represent application
 * state at the time of request; as such, no methods are provided to allow
 * modification of those values. The other values provide such methods, as they
 * can be restored from $_SERVER or the request body, and may need treatment
 * during the application (e.g., body parameters may be deserialized based on
 * content type).
 *
 * Additionally, this interface recognizes the utility of introspecting a
 * request to derive and match additional parameters (e.g., via URI path
 * matching, decrypting cookie values, deserializing non-form-encoded body
 * content, matching authorization headers to users, etc). These parameters
 * are stored in an "attributes" property.
 *
 * HTTP 请求是被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套机制，在内部保
 * 持好原有的内容，然后把修改状态后的，新的 HTTP 请求实例返回。
 */
interface ServerRequestInterface extends RequestInterface
{
    /**
     * Retrieve server parameters.
     *
```

```
     * Retrieves data related to the incoming request environment,
     * typically derived from PHP's $_SERVER superglobal. The data IS NOT
     * REQUIRED to originate from $_SERVER.
     *
     * @return array
     */
    public function getServerParams();

    /**
     * Retrieve cookies.
     *
     * Retrieves cookies sent by the client to the server.
     *
     * The data MUST be compatible with the structure of the $_COOKIE
     * superglobal.
     *
     * @return array
     */
    public function getCookieParams();

    /**
     * Return an instance with the specified cookies.
     *
     * The data IS NOT REQUIRED to come from the $_COOKIE superglobal, but MUST
     * be compatible with the structure of $_COOKIE. Typically, this data will
     * be injected at instantiation.
     *
     * This method MUST NOT update the related Cookie header of the request
     * instance, nor related values in the server params.
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param array $cookies Array of key/value pairs representing cookies.
     * @return self
     */
    public function withCookieParams(array $cookies);

    /**
     * Retrieve query string arguments.
     *
```

```
     * Retrieves the deserialized query string arguments, if any.
     *
     * Note: the query params might not be in sync with the URI or server
     * params. If you need to ensure you are only getting the original
     * values, you may need to parse the query string from `getUri()->getQuery()`
     * or from the `QUERY_STRING` server param.
     *
     * @return array
     */
    public function getQueryParams();

    /**
     * Return an instance with the specified query string arguments.
     *
     * These values SHOULD remain immutable over the course of the incoming
     * request. They MAY be injected during instantiation, such as from PHP's
     * $_GET superglobal, or MAY be derived from some other value such as the
     * URI. In cases where the arguments are parsed from the URI, the data
     * MUST be compatible with what PHP's parse_str() would return for
     * purposes of how duplicate query parameters are handled, and how nested
     * sets are handled.
     *
     * Setting query string arguments MUST NOT change the URI stored by the
     * request, nor the values in the server params.
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param array $query Array of query string arguments, typically from
     *     $_GET.
     * @return self
     */
    public function withQueryParams(array $query);

    /**
     * Retrieve normalized file upload data.
     *
     * This method returns upload metadata in a normalized tree, with each leaf
     * an instance of Psr\Http\Message\UploadedFileInterface.
     *
     * These values MAY be prepared from $_FILES or the message body during
```

```
     * instantiation, or MAY be injected via withUploadedFiles().
     *
     * @return array An array tree of UploadedFileInterface instances; an empty
     *    array MUST be returned if no data is present.
     */
    public function getUploadedFiles();

    /**
     * Create a new instance with the specified uploaded files.
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param array An array tree of UploadedFileInterface instances.
     * @return self
     * @throws \InvalidArgumentException if an invalid structure is provided.
     */
    public function withUploadedFiles(array $uploadedFiles);

    /**
     * Retrieve any parameters provided in the request body.
     *
     * If the request Content-Type is either application/x-www-form-urlencoded
     * or multipart/form-data, and the request method is POST, this method MUST
     * return the contents of $_POST.
     *
     * Otherwise, this method may return any results of deserializing
     * the request body content; as parsing returns structured content, the
     * potential types MUST be arrays or objects only. A null value indicates
     * the absence of body content.
     *
     * @return null|array|object The deserialized body parameters, if any.
     *    These will typically be an array or object.
     */
    public function getParsedBody();

    /**
     * Return an instance with the specified body parameters.
     *
     * These MAY be injected during instantiation.
     *
```

```
     * If the request Content-Type is either application/x-www-form-urlencoded
     * or multipart/form-data, and the request method is POST, use this method
     * ONLY to inject the contents of $_POST.
     *
     * The data IS NOT REQUIRED to come from $_POST, but MUST be the results of
     * deserializing the request body content. Deserialization/parsing returns
     * structured data, and, as such, this method ONLY accepts arrays or objects,
     * or a null value if nothing was available to parse.
     *
     * As an example, if content negotiation determines that the request data
     * is a JSON payload, this method could be used to create a request
     * instance with the deserialized parameters.
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @param null|array|object $data The deserialized body data. This will
     *     typically be in an array or object.
     * @return self
     * @throws \InvalidArgumentException if an unsupported argument type is
     *     provided.
     */
    public function withParsedBody($data);

    /**
     * Retrieve attributes derived from the request.
     *
     * The request "attributes" may be used to allow injection of any
     * parameters derived from the request: e.g., the results of path
     * match operations; the results of decrypting cookies; the results of
     * deserializing non-form-encoded message bodies; etc. Attributes
     * will be application and request specific, and CAN be mutable.
     *
     * @return mixed[] Attributes derived from the request.
     */
    public function getAttributes();

    /**
     * Retrieve a single derived request attribute.
     *
     * Retrieves a single derived request attribute as described in
```

```
     * getAttributes(). If the attribute has not been previously set, returns
     * the default value as provided.
     *
     * This method obviates the need for a hasAttribute() method, as it allows
     * specifying a default value to return if the attribute is not found.
     *
     * @see getAttributes()
     * @param string $name The attribute name.
     * @param mixed $default Default value to return if the attribute does not exist.
     * @return mixed
     */
    public function getAttribute($name, $default = null);

    /**
     * Return an instance with the specified derived request attribute.
     *
     * This method allows setting a single derived request attribute as
     * described in getAttributes().
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @see getAttributes()
     * @param string $name The attribute name.
     * @param mixed $value The value of the attribute.
     * @return self
     */
    public function withAttribute($name, $value);

    /**
     * Return an instance that removes the specified derived request attribute.
     *
     * This method allows removing a single derived request attribute as
     * described in getAttributes().
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @see getAttributes()
     * @param string $name The attribute name.
     * @return self
```

```
     */
    public function withoutAttribute($name);
}
```

143.7.4 Psr\Http\Message\ResponseInterface

```
<?php
namespace Psr\Http\Message;

/**
 * Representation of an outgoing, server-side response.
 *
 * Per the HTTP specification, this interface includes properties for
 * each of the following:
 *
 * - Protocol version
 * - Status code and reason phrase
 * - Headers
 * - Message body
 *
 * Responses are considered immutable; all methods that might change state MUST
 * be implemented such that they retain the internal state of the current
 * message and return an instance that contains the changed state.
 *
 * HTTP 响应是被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套机制，在内部保
 * 持好原有的内容，然后把修改状态后的，新的 HTTP 响应实例返回。
 */
interface ResponseInterface extends MessageInterface
{
    /**
     * Gets the response status code.
     *
     * The status code is a 3-digit integer result code of the server's attempt
     * to understand and satisfy the request.
     *
     * @return int Status code.
     */
    public function getStatusCode();

    /**
     * Return an instance with the specified status code and, optionally, reason phrase.
```

```
     *
     * If no reason phrase is specified, implementations MAY choose to default
     * to the RFC 7231 or IANA recommended reason phrase for the response's
     * status code.
     *
     * 此方法在实现的时候，**必须** 保留原有的不可修改的 HTTP 消息实例，然后返回
     * 一个新的修改过的 HTTP 消息实例。
     *
     * @see http://tools.ietf.org/html/rfc7231#section-6
     * @see http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
     * @param int $code The 3-digit integer result code to set.
     * @param string $reasonPhrase The reason phrase to use with the
     *     provided status code; if none is provided, implementations MAY
     *     use the defaults as suggested in the HTTP specification.
     * @return self
     * @throws \InvalidArgumentException For invalid status code arguments.
     */
    public function withStatus($code, $reasonPhrase = '');

    /**
     * Gets the response reason phrase associated with the status code.
     *
     * Because a reason phrase is not a required element in a response
     * status line, the reason phrase value MAY be empty. Implementations MAY
     * choose to return the default RFC 7231 recommended reason phrase (or those
     * listed in the IANA HTTP Status Code Registry) for the response's
     * status code.
     *
     * @see http://tools.ietf.org/html/rfc7231#section-6
     * @see http://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml
     * @return string Reason phrase; must return an empty string if none present.
     */
    public function getReasonPhrase();
}
```

143.7.5 Psr\Http\Message\StreamInterface

```
<?php
namespace Psr\Http\Message;

/**
```

```
 * Describes a data stream.
 *
 * Typically, an instance will wrap a PHP stream; this interface provides
 * a wrapper around the most common operations, including serialization of
 * the entire stream to a string.
 */
interface StreamInterface
{
    /**
     * Reads all data from the stream into a string, from the beginning to end.
     *
     * This method MUST attempt to seek to the beginning of the stream before
     * reading data and read the stream until the end is reached.
     *
     * Warning: This could attempt to load a large amount of data into memory.
     *
     * This method MUST NOT raise an exception in order to conform with PHP's
     * string casting operations.
     *
     * @see http://php.net/manual/en/language.oop5.magic.php#object.tostring
     * @return string
     */
    public function __toString();

    /**
     * Closes the stream and any underlying resources.
     *
     * @return void
     */
    public function close();

    /**
     * Separates any underlying resources from the stream.
     *
     * After the stream has been detached, the stream is in an unusable state.
     *
     * @return resource|null Underlying PHP stream, if any
     */
    public function detach();

    /**
```

```
     * Get the size of the stream if known.
     *
     * @return int|null Returns the size in bytes if known, or null if unknown.
     */
    public function getSize();

    /**
     * Returns the current position of the file read/write pointer
     *
     * @return int Position of the file pointer
     * @throws \RuntimeException on error.
     */
    public function tell();

    /**
     * Returns true if the stream is at the end of the stream.
     *
     * @return bool
     */
    public function eof();

    /**
     * Returns whether or not the stream is seekable.
     *
     * @return bool
     */
    public function isSeekable();

    /**
     * Seek to a position in the stream.
     *
     * @see http://www.php.net/manual/en/function.fseek.php
     * @param int $offset Stream offset
     * @param int $whence Specifies how the cursor position will be calculated
     *     based on the seek offset. Valid values are identical to the built-in
     *     PHP $whence values for `fseek()`. SEEK_SET: Set position equal to
     *     offset bytes SEEK_CUR: Set position to current location plus offset
     *     SEEK_END: Set position to end-of-stream plus offset.
     * @throws \RuntimeException on failure.
     */
    public function seek($offset, $whence = SEEK_SET);
```

```
/**
 * Seek to the beginning of the stream.
 *
 * If the stream is not seekable, this method will raise an exception;
 * otherwise, it will perform a seek(0).
 *
 * @see seek()
 * @see http://www.php.net/manual/en/function.fseek.php
 * @throws \RuntimeException on failure.
 */
public function rewind();

/**
 * Returns whether or not the stream is writable.
 *
 * @return bool
 */
public function isWritable();

/**
 * Write data to the stream.
 *
 * @param string $string The string that is to be written.
 * @return int Returns the number of bytes written to the stream.
 * @throws \RuntimeException on failure.
 */
public function write($string);

/**
 * Returns whether or not the stream is readable.
 *
 * @return bool
 */
public function isReadable();

/**
 * Read data from the stream.
 *
 * @param int $length Read up to $length bytes from the object and return
 *     them. Fewer than $length bytes may be returned if underlying stream
```

```
     *     call returns fewer bytes.
     * @return string Returns the data read from the stream, or an empty string
     *     if no bytes are available.
     * @throws \RuntimeException if an error occurs.
     */
    public function read($length);

    /**
     * Returns the remaining contents in a string
     *
     * @return string
     * @throws \RuntimeException if unable to read.
     * @throws \RuntimeException if error occurs while reading.
     */
    public function getContents();

    /**
     * Get stream metadata as an associative array or retrieve a specific key.
     *
     * The keys returned are identical to the keys returned from PHP's
     * stream_get_meta_data() function.
     *
     * @see http://php.net/manual/en/function.stream-get-meta-data.php
     * @param string $key Specific metadata to retrieve.
     * @return array|mixed|null Returns an associative array if no key is
     *     provided. Returns a specific key value if a key is provided and the
     *     value is found, or null if the key is not found.
     */
    public function getMetadata($key = null);
}
```

143.7.6 Psr\Http\Message\UriInterface

```
<?php
namespace Psr\Http\Message;

/**
 * URI 数据对象。
 *
 * 此接口按照 RFC 3986 来构建 HTTP URI，提供了一些通用的操作，你可以自由的对此接口
 * 进行扩展。你可以使用此 URI 接口来做 HTTP 相关的操作，也可以使用此接口做任何 URI
```

```
 * 相关的操作。
 *
 * 此接口的实例化对象被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套机制，在内部保
 * 持好原有的内容，然后把修改状态后的，新的实例返回。
 *
 * @see http://tools.ietf.org/html/rfc3986 (URI 通用标准规范)
 */
interface UriInterface
{
    /**
     * 从 URI 中取出 scheme。
     *
     * 如果不存在 Scheme，此方法 **必须** 返回空字符串。
     *
     * 返回的数据 **必须** 是小写字母，遵照 RFC 3986 规范 3.1 章节。
     *
     * 最后部分的 ":" 字串不属于 Scheme，**一定不可** 作为返回数据的一部分。
     *
     * @see https://tools.ietf.org/html/rfc3986#section-3.1
     * @return string URI scheme 的值
     */
    public function getScheme();

    /**
     * 返回 URI 授权信息。
     *
     * 如果没有 URI 信息的话，**必须** 返回一个空数组。
     *
     * URI 的授权信息语法是：
     *
     * <pre>
     * [user-info@]host[:port]
     * </pre>
     *
     * 如果端口部分没有设置，或者端口不是标准端口，**一定不可** 包含在返回值内。
     *
     * @see https://tools.ietf.org/html/rfc3986#section-3.2
     * @return string URI 授权信息，格式为："[user-info@]host[:port]"
     */
    public function getAuthority();
```

```
/**
 * 从 URI 中获取用户信息。
 *
 * 如果不存在用户信息，此方法 **必须** 返回一个空字符串。
 *
 * 用户信息后面跟着的 "@" 字符，不是用户信息里面的一部分，**一定不可** 在返回值里
 * 出现。
 *
 * @return string URI 的用户信息，格式："username[:password]"
 */
public function getUserInfo();

/**
 * 从 URI 信息中获取 HOST 值。
 *
 * 如果 URI 中没有此值，**必须** 返回空字符串。
 *
 * 返回的数据 **必须** 是小写字母，遵照 RFC 3986 规范 3.2.2 章节。
 *
 * @see http://tools.ietf.org/html/rfc3986#section-3.2.2
 * @return string URI 信息中的 HOST 值。
 */
public function getHost();

/**
 * 从 URI 信息中获取端口信息。
 *
 * 如果端口信息是与当前 Scheme 的标准端口不匹配的话，就使用整数值的格式返回，如果是一
 * 样的话，**必须** 返回 `null` 值。
 *
 * 如果存在端口信息，都是不存在 scheme 信息的话，**必须** 返回 `null` 值。
 *
 * 如果不存在端口数据，但是 scheme 数据存在的话，**可以** 返回 scheme 对应的
 * 标准端口，但是 **应该** 返回 `null`。
 *
 * @return null|int 从 URI 信息中的端口信息。
 */
public function getPort();

/**
```

```
     * 从 URI 信息中获取路径。
     *
     *  The path can either be empty or absolute (starting with a slash) or
     * rootless (not starting with a slash). Implementations MUST support all
     * three syntaxes.
     *
     * Normally, the empty path "" and absolute path "/" are considered equal as
     * defined in RFC 7230 Section 2.7.3. But this method MUST NOT automatically
     * do this normalization because in contexts with a trimmed base path, e.g.
     * the front controller, this difference becomes significant. It's the task
     * of the user to handle both "" and "/".
     *
     * The value returned MUST be percent-encoded, but MUST NOT double-encode
     * any characters. To determine what characters to encode, please refer to
     * RFC 3986, Sections 2 and 3.3.
     *
     * As an example, if the value should include a slash ("/") not intended as
     * delimiter between path segments, that value MUST be passed in encoded
     * form (e.g., "%2F") to the instance.
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.3
     * @return string The URI path.
     */
    public function getPath();

    /**
     * Retrieve the query string of the URI.
     *
     * If no query string is present, this method MUST return an empty string.
     *
     * The leading "?" character is not part of the query and MUST NOT be
     * added.
     *
     * The value returned MUST be percent-encoded, but MUST NOT double-encode
     * any characters. To determine what characters to encode, please refer to
     * RFC 3986, Sections 2 and 3.4.
     *
     * As an example, if a value in a key/value pair of the query string should
     * include an ampersand ("&") not intended as a delimiter between values,
     * that value MUST be passed in encoded form (e.g., "%26") to the instance.
```

```
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.4
     * @return string The URI query string.
     */
    public function getQuery();

    /**
     * Retrieve the fragment component of the URI.
     *
     * If no fragment is present, this method MUST return an empty string.
     *
     * The leading "#" character is not part of the fragment and MUST NOT be
     * added.
     *
     * The value returned MUST be percent-encoded, but MUST NOT double-encode
     * any characters. To determine what characters to encode, please refer to
     * RFC 3986, Sections 2 and 3.5.
     *
     * @see https://tools.ietf.org/html/rfc3986#section-2
     * @see https://tools.ietf.org/html/rfc3986#section-3.5
     * @return string The URI fragment.
     */
    public function getFragment();

    /**
     * Return an instance with the specified scheme.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified scheme.
     *
     * Implementations MUST support the schemes "http" and "https" case
     * insensitively, and MAY accommodate other schemes if required.
     *
     * An empty scheme is equivalent to removing the scheme.
     *
     * @param string $scheme The scheme to use with the new instance.
     * @return self A new instance with the specified scheme.
     * @throws \InvalidArgumentException for invalid schemes.
     * @throws \InvalidArgumentException for unsupported schemes.
     */
```

```
    public function withScheme($scheme);

    /**
     * Return an instance with the specified user information.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified user information.
     *
     * Password is optional, but the user information MUST include the
     * user; an empty string for the user is equivalent to removing user
     * information.
     *
     * @param string $user The user name to use for authority.
     * @param null|string $password The password associated with $user.
     * @return self A new instance with the specified user information.
     */
    public function withUserInfo($user, $password = null);

    /**
     * Return an instance with the specified host.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified host.
     *
     * An empty host value is equivalent to removing the host.
     *
     * @param string $host The hostname to use with the new instance.
     * @return self A new instance with the specified host.
     * @throws \InvalidArgumentException for invalid hostnames.
     */
    public function withHost($host);

    /**
     * Return an instance with the specified port.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified port.
     *
     * Implementations MUST raise an exception for ports outside the
     * established TCP and UDP port ranges.
     *
```

```
     * A null value provided for the port is equivalent to removing the port
     * information.
     *
     * @param null|int $port The port to use with the new instance; a null value
     *    removes the port information.
     * @return self A new instance with the specified port.
     * @throws \InvalidArgumentException for invalid ports.
     */
    public function withPort($port);

    /**
     * Return an instance with the specified path.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified path.
     *
     * The path can either be empty or absolute (starting with a slash) or
     * rootless (not starting with a slash). Implementations MUST support all
     * three syntaxes.
     *
     * If an HTTP path is intended to be host-relative rather than path-relative
     * then it must begin with a slash ("/"). HTTP paths not starting with a slash
     * are assumed to be relative to some base path known to the application or
     * consumer.
     *
     * Users can provide both encoded and decoded path characters.
     * Implementations ensure the correct encoding as outlined in getPath().
     *
     * @param string $path The path to use with the new instance.
     * @return self A new instance with the specified path.
     * @throws \InvalidArgumentException for invalid paths.
     */
    public function withPath($path);

    /**
     * Return an instance with the specified query string.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified query string.
     *
     * Users can provide both encoded and decoded query characters.
```

```
     * Implementations ensure the correct encoding as outlined in getQuery().
     *
     * An empty query string value is equivalent to removing the query string.
     *
     * @param string $query The query string to use with the new instance.
     * @return self A new instance with the specified query string.
     * @throws \InvalidArgumentException for invalid query strings.
     */
    public function withQuery($query);

    /**
     * Return an instance with the specified URI fragment.
     *
     * This method MUST retain the state of the current instance, and return
     * an instance that contains the specified URI fragment.
     *
     * Users can provide both encoded and decoded fragment characters.
     * Implementations ensure the correct encoding as outlined in getFragment().
     *
     * An empty fragment value is equivalent to removing the fragment.
     *
     * @param string $fragment The fragment to use with the new instance.
     * @return self A new instance with the specified fragment.
     */
    public function withFragment($fragment);

    /**
     * Return the string representation as a URI reference.
     *
     * Depending on which components of the URI are present, the resulting
     * string is either a full URI or relative reference according to RFC 3986,
     * Section 4.1. The method concatenates the various components of the URI,
     * using the appropriate delimiters:
     *
     * - If a scheme is present, it MUST be suffixed by ":".
     * - If an authority is present, it MUST be prefixed by "//".
     * - The path can be concatenated without delimiters. But there are two
     *   cases where the path has to be adjusted to make the URI reference
     *   valid as PHP does not allow to throw an exception in __toString():
     *     - If the path is rootless and an authority is present, the path MUST
     *       be prefixed by "/".
```

```
     *    - If the path is starting with more than one "/" and no authority is
     *      present, the starting slashes MUST be reduced to one.
     * - If a query is present, it MUST be prefixed by "?".
     * - If a fragment is present, it MUST be prefixed by "#".
     *
     * @see http://tools.ietf.org/html/rfc3986#section-4.1
     * @return string
     */
    public function __toString();
}
```

143.7.7 Psr\Http\Message\UploadedFileInterface

```
<?php
namespace Psr\Http\Message;

/**
 * 通过 HTTP 请求上传的一个文件内容。
 *
 * 此接口的实例是被视为无法修改的，所有能修改状态的方法，都 **必须** 有一套机制，在内部保
 * 持好原有的内容，然后把修改状态后的，新的实例返回。
 */
interface UploadedFileInterface
{
    /**
     * 获取上传文件的数据流。
     *
     * 此方法必须返回一个 `StreamInterface` 实例，此方法的目的在于允许 PHP 对获取到的数
     * 据流直接操作，如 stream_copy_to_stream() 。
     *
     * 如果在调用此方法之前调用了 `moveTo()` 方法，此方法 **必须** 抛出异常。
     *
     * @return StreamInterface 上传文件的数据流
     * @throws \RuntimeException 没有数据流的情形下。
     * @throws \RuntimeException 无法创建数据流。
     */
    public function getStream();

    /**
     * 把上传的文件移动到新目录。
     *
```

```
     * 此方法保证能同时在 `SAPI` 和 `non-SAPI` 环境下使用。实现类库 **必须** 判断
     * 当前处在什么环境下，并且使用合适的方法来处理，如 move_uploaded_file()，rename()
     * 或者数据流操作。
     *
     * $targetPath 可以是相对路径，也可以是绝对路径，使用 rename() 解析起来应该是一样的。
     *
     * 当这一次完成后，原来的文件 **必须** 会被移除。
     *
     * 如果此方法被调用多次，一次以后的其他调用，都要抛出异常。
     *
     * 如果在 SAPI 环境下的话，$_FILES 内有值，当使用 moveTo()，is_uploaded_file()
     * 和 move_uploaded_file() 方法来移动文件时 **应该** 确保权限和上传状态的准确性。
     *
     * 如果你希望操作数据流的话，请使用 `getStream()` 方法，因为在 SAPI 场景下，无法
     * 保证书写入数据流目标。
     *
     * @see http://php.net/is_uploaded_file
     * @see http://php.net/move_uploaded_file
     * @param string $targetPath 目标文件路径。
     * @throws \InvalidArgumentException 参数有问题时抛出异常。
     * @throws \RuntimeException 发生任何错误，都抛出此异常。
     * @throws \RuntimeException 多次运行，也抛出此异常。
     */
    public function moveTo($targetPath);

    /**
     * 获取文件大小。
     *
     * 实现类库 **应该** 优先使用 $_FILES 里的 `size` 数值。
     *
     * @return int|null 以 bytes 为单位，或者 null 未知的情况下。
     */
    public function getSize();

    /**
     * 获取上传文件时出现的错误。
     *
     * 返回值 **必须** 是 PHP 的 UPLOAD_ERR_XXX 常量。
     *
     * 如果文件上传成功，此方法 **必须** 返回 UPLOAD_ERR_OK。
     *
```

```
     * 实现类库 **必须** 返回 $_FILES 数组中的 `error` 值。
     *
     * @see http://php.net/manual/en/features.file-upload.errors.php
     * @return int PHP 的 UPLOAD_ERR_XXX 常量。
     */
    public function getError();

    /**
     * 获取客户端上传的文件的名称。
     *
     * 永远不要信任此方法返回的数据，客户端有可能发送了一个恶意的文件名来攻击你的程序。
     *
     * 实现类库 **应该** 返回存储在 $_FILES 数组中 `name` 的值。
     *
     * @return string|null 用户上传的名字，或者 null 如果没有此值。
     */
    public function getClientFilename();

    /**
     * 客户端提交的文件类型。
     *
     * 永远不要信任此方法返回的数据，客户端有可能发送了一个恶意的文件类型名称来攻击你的程序。
     *
     * 实现类库 **应该** 返回存储在 $_FILES 数组中 `type` 的值。
     *
     * @return string|null 用户上传的类型，或者 null 如果没有此值。
     */
    public function getClientMediaType();
}
```

Chapter 144

PSR-8

144.1 Overview

This standard establishes a common way for objects to express mutual appreciation and support by hugging. This allows objects to support each other in a constructive fashion, furthering cooperation between different PHP projects.

This specification defines two interfaces, \Psr\Hug\Huggable and \Psr\Hug\GroupHuggable.

144.1.1 Huggable Objects

1. A Huggable object expresses affection and support for another object by invoking its hug() method, passing $this as the first parameter.
2. An object whose hug() method is invoked MUST hug() the calling object back at least once.
3. Two objects that are engaged in a hug MAY continue to hug each other back for any number of iterations. However, every huggable object MUST have a termination condition that will prevent an infinite loop. For example, an object MAY be configured to only allow up to 3 mutual hugs, after which it will break the hug chain and return.
4. An object MAY take additional actions, including modifying state, when hugged. A common example is to increment an internal happiness or satisfaction counter.

144.1.2 GroupHuggable objects

1. An object may optionally implement GroupHuggable to indicate that it is able to support and affirm multiple objects at once.

144.2 Interface

144.2.1 Psr\Hug\HuggableInterface

```
namespace Psr\Hug;

/**
 * Defines a huggable object.
 *
 * A huggable object expresses mutual affection with another huggable object.
 */
interface Huggable
{

    /**
     * Hugs this object.
     *
     * All hugs are mutual. An object that is hugged MUST in turn hug the other
     * object back by calling hug() on the first parameter. All objects MUST
     * implement a mechanism to prevent an infinite loop of hugging.
     *
     * @param Huggable $h
     *   The object that is hugging this object.
     */
    public function hug(Huggable $h);
}
```

144.2.2 Psr\Hug\GroupHuggable

```
namespace Psr\Hug;

/**
 * Defines a huggable object.
 *
 * A huggable object expresses mutual affection with another huggable object.
 */
interface GroupHuggable extends Huggable
{

  /**
   * Hugs a series of huggable objects.
```

```
   *
   * When called, this object MUST invoke the hug() method of every object
   * provided. The order of the collection is not significant, and this object
   * MAY hug each of the objects in any order provided that all are hugged.
   *
   * @param $huggables
   *   An array or iterator of objects implementing the Huggable interface.
   */
  public function groupHug($huggables);
}
```

Chapter 145

PSR-9

145.1 Overview

There are two aspects with dealing with security issues: One is the process by which security issues are reported and fixed in projects, the other is how the general public is informed about the issues and any remedies available. While PSR-9 addresses the former, this PSR, ie. PSR-10, deals with the later. So the goal of PSR-10 is to define how security issues are disclosed to the public and what format such disclosures should follow. Especially today where PHP developers are sharing code across projects more than ever, this PSR aims to ease the challenges in keeping an overview of security issues in all dependencies and the steps required to address them.

The goal of this PSR is to give project leads a clearly defined approach to enabling end users to discover security disclosures using a clearly defined structured format for these disclosures.

145.2 Disclosure Discovery

Every project MUST provide a link to its security vulnerability database in an obvious place. Ideally this should be on the root page of the main domain of the given project. This MAY be a sub-domain in case it is a sub-project of a larger initiative. If the project has a dedicated page for its disclosure process discovery then this is also considered a good place for this link. The link MAY use the custom link relation php-vuln-disclosures, ie. for example <link rel=”php-vuln-disclosures” href=”http://example.org/disclosures”/>.

Note that projects MAY choose to host their disclosure files on a domain other than their main project page. It is RECOMMENDED to not store the disclosures in a VCS as this can lead to the confusions about which branch is the relevant branch. If a VCS is used then additional steps SHOULD be taken to clearly document to users which branch contains all vulnerabilities for all versions. If

necessary projects MAY however split vulnerability disclosure files by major version number. In this case again this SHOULD be clearly documented.

145.3 Disclosure Format

The disclosure format is based on Atom [1], which in turn is based on XML. It leverages the "The Common Vulnerability Reporting Framework (CVRF) v1.1" [2]. Specifically it leverages its dictionary [3] as its base terminology.

TODO: Should we also provide a JSON serialization to lower the bar for projects. Aggregation services can then spring up to provide an Atom representation of these disclosures in JSON format.

The Atom extensions [4] allow a structured description of the vulnerability to enable automated tools to determine if installed is likely affected by the vulnerability. However human readability is considered highly important and as such not the full CVRF is used.

TODO: Review the Atom format and the supplied XSD

Note that for each vulnerability only a single entry MUST be created. In case any information changes the original file MUST be updated along with the last update field.

Any disclosure uses entryType using the following tags from the Atom namespace (required tags are labeled with "MUST"):

- title (short description of the vulnerability and affected versions, MUST)
- summary (description of the vulnerability)
- author (contact information, MUST)
- published (initial publication date, MUST)
- updated (date of the last update)
- link (to reference more information)
- id (project specific vulnerability id)

In addition the following tags are added:

- reported (initial report date)
- reportedBy (contact information for the persons or entity that initially reported the vulnerability)
- resolvedBy (contact information for the persons or entity that resolved the vulnerability)
- name (name of the product, MUST)
- cve (unique CVE ID)
- cwe (unique CWE ID)
- severity (low, medium high)
- affected (version(s) using composer syntax [5])
- status (open, in progress, disputed, completed, MUST)

- remediation (textual description for how to fix an affected system)
- remediationType (workaround, mitigation, vendor fix, none available, will not fix)
- remediationLink (URL to give additional information for remediation)

Chapter 146

PSR-10

146.1 Overview

There are two aspects with dealing with security issues: One is the process by which security issues are reported and fixed in projects, the other is how the general public is informed about the issues and any remedies available. While PSR-10 addresses the later, this PSR, ie. PSR-9, deals with the former. So the goal of PSR-9 is to define the process by which security researchers and report security vulnerabilities to projects. It is important that when security vulnerabilities are found that researchers have an easy channel to the projects in question allowing them to disclose the issue to a controlled group of people.

The goal of this PSR is to give researchers, project leads, upstream project leads and end users a defined and structured process for disclosing security vulnerabilities.

146.2 Security Disclosure Process Discovery

Every project MUST provide a link to its security disclosure process in an obvious place. Ideally this should be on the root page the main domain of the given project. This MAY be a sub-domain in case it is a sub-project of a larger initiative. The link MAY use the custom link relation php-vuln-reporting, ie. for example <link rel="php-vuln-reporting" href="http://example.org/security"/>.

Projects SHOULD ideally make the location prominent itself by either creating a dedicated sub-domain like http://security.example.org or by making it a top level directory like http://example.org/security. Alternatively projects MAY also simply reference this document, ie. PSR-9. By referencing PSR-9 a project basically states that they follow the default procedures as noted in the section "Default Procedures" towards the end of this document. Projects MUST list the variables noted at the start of that section in this reference (ie. project name, project domain, etc.). Projects MAY choose to list any

part of the procedures that is not a MUST which they choose to omit.

Note that projects MAY not have a dedicated domain. For example a project hosted on Github, Bitbucket or other service should still ensure that the process is referenced on the landing page, ie. for example http://github.com/example/somelib should ensure that the default branch has a README file which references the procedures used so that it is automatically displayed.

If necessary projects MAY have different disclosure process for different major version number. In this case one URL MUST be provided for each major version. In the case a major version is no longer receiving security fixes, instead of an URL a project MAY opt to instead simply note that the version is no longer receiving security fixes.

146.3 Security Disclosure Process

Every project MUST provide an email address in their security disclosure process description as the contact email address. Projects SHALL NOT use contact forms.

146.4 Default Procedures

project name denotes the name on which the project uses to identify itself.
project domain denotes the main (sub)domain on which the project relies.

If not specified otherwise, the contact email address is security@[project domain].

Chapter 147

PSR-11

147.1 Overview

This document describes a common interface for dependency injection containers.

The goal set by ContainerInterface is to standardize how frameworks and libraries make use of a container to obtain objects and parameters (called entries in the rest of this document).

The word implementor in this document is to be interpreted as someone implementing the ContainerInterface in a dependency injection-related library or framework. Users of dependency injections containers (DIC) are referred to as user.

147.2 Specification

147.2.1 Basics

The Psr\Container\ContainerInterface exposes two methods : get and has.

- `get` takes one mandatory parameter: an entry identifier. It MUST be a string. A call to get can return anything (a mixed value), or throws an exception if the identifier is not known to the container. Two successive calls to get with the same identifier SHOULD return the same value. However, depending on the implementor design and/or user configuration, different values might be returned, so user SHOULD NOT rely on getting the same value on 2 successive calls. While ContainerInterface only defines one mandatory parameter in get(), implementations MAY accept additional optional parameters.
- `has` takes one unique parameter: an entry identifier. It MUST return true if an entry identifier is known to the container and false if it is not. has($id) returning true does not mean that get($id) will not throw an exception. It does however mean that get($id) will not throw a

NotFoundException.

147.2.2 Exceptions

Exceptions directly thrown by the container MUST implement the Psr\Container\Exception\ContainerException

A call to the get method with a non-existing id SHOULD throw a Psr\Container\Exception\NotFoundExceptionIn

Users SHOULD NOT pass a container into an object so that the object can retrieve its own dependencies. This means the container is used as a Service Locator which is a pattern that is generally discouraged.

147.3 Delegate

This section describes an additional feature that MAY be added to a container. Containers are not required to implement the delegate lookup to respect the ContainerInterface.

The goal of the delegate lookup feature is to allow several containers to share entries. Containers implementing this feature can perform dependency lookups in other containers.

Containers implementing this feature will offer a greater lever of interoperability with other containers. Implementation of this feature is therefore RECOMMENDED.

A container implementing this feature:

- MUST implement the ContainerInterface
- MUST provide a way to register a delegate container (using a constructor parameter, or a setter, or any possible way). The delegate container MUST implement the ContainerInterface.

When a container is configured to use a delegate container for dependencies:

- Calls to the `get` method should only return an entry if the entry is part of the container. If the entry is not part of the container, an exception should be thrown (as requested by the ContainerInterface).
- Calls to the `has` method should only return true if the entry is part of the container. If the entry is not part of the container, false should be returned.
- If the fetched entry has dependencies, instead of performing the dependency lookup in the container, the lookup is performed on the delegate container.

By default, the lookup SHOULD be performed on the delegate container only, not on the container itself.

It is however allowed for containers to provide exception cases for special entries, and a way to lookup into the same container (or another container) instead of the delegate container.

147.4 Package

The interfaces and classes described as well as relevant exception are provided as part of the psr/container package. (still to-be-created)

147.5 Interface

147.5.1 Psr\Container\ContainerInterface

```
<?php
namespace Psr\Container;

use Psr\Container\Exception\ContainerExceptionInterface;
use Psr\Container\Exception\NotFoundExceptionInterface;

/**
 * Describes the interface of a container that exposes methods to read its entries.
 */
interface ContainerInterface
{
    /**
     * Finds an entry of the container by its identifier and returns it.
     *
     * @param string $id Identifier of the entry to look for.
     *
     * @throws NotFoundExceptionInterface No entry was found for this identifier.
     * @throws ContainerExceptionInterface Error while retrieving the entry.
     *
     * @return mixed Entry.
     */
    public function get($id);

    /**
     * Returns true if the container can return an entry for the given identifier.
     * Returns false otherwise.
     *
     * `has($id)` returning true does not mean that `get($id)` will not throw an
         exception.
     * It does however mean that `get($id)` will not throw a `NotFoundException`.
     *
     * @param string $id Identifier of the entry to look for.
```

```
     *
     * @return boolean
     */
    public function has($id);
}
```

147.5.2 Psr\Container\Exception\ContainerExceptionInterface

```
<?php
namespace Psr\Container\Exception;

/**
 * Base interface representing a generic exception in a container.
 */
interface ContainerExceptionInterface
{
}
```

147.5.3 Psr\Container\Exception\NotFoundExceptionInterface

```
<?php
namespace Psr\Container\Exception;

/**
 * No entry was found in the container.
 */
interface NotFoundExceptionInterface extends ContainerExceptionInterface
{
}
```

Chapter 148

PSR-12

148.1 Overview

This specification extends, expands and replaces PSR-2, the coding style guide and requires adherance to PSR-1, the basic coding standard.

Like PSR-2, the intent of this specification is to reduce cognitive friction when scanning code from different authors. It does so by enumerating a shared set of rules and expectations about how to format PHP code. This PSR seeks to provide a set way that coding style tools can implement, projects can declare adherence to and developers can easily relate to between different projects. When various authors collaborate across multiple projects, it helps to have one set of guidelines to be used among all those projects. Thus, the benefit of this guide is not in the rules themselves but the sharing of those rules.

PSR-2 was accepted in 2012 and since then a number of changes have been made to PHP which have implications for coding style guidelines. Whilst PSR-2 is very comprehensive of PHP functionality that existed at the time of writing, new functionality is very open to interpretation. This PSR therefore seeks to clarify the content of PSR-2 in a more modern context with new functionality available, and make the errata to PSR-2 binding.

Throughout this document, any instructions MAY be ignored if they do not exist in versions of PHP supported by your project.

This example encompasses some of the rules below as a quick overview:

```
<?php
declare(strict_types=1);

namespace Vendor\Package;
```

```
use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\Namespace\ClassD as D;

use function Vendor\Package\{functionA, functionB, functionC};
use const Vendor\Package\{ConstantA, ConstantB, ConstantC};

class Foo extends Bar implements FooInterface
{
    public function sampleFunction(int $a, int $b = null): array
    {
        if ($a === $b) {
            bar();
        } elseif ($a > $b) {
            $foo->bar($arg1);
        } else {
            BazClass::bar($arg2, $arg3);
        }
    }

    final public static function bar()
    {
        // method body
    }
}
```

148.2 Basics

Code MUST follow all rules outlined in PSR-1.

The term 'StudlyCaps' in PSR-1 MUST be interpreted as PascalCase where the first letter of each word is capitalised including the very first letter.

148.3 Files

All PHP files MUST use the Unix LF (linefeed) line ending.

All PHP files MUST end with a single line, containing only a single newline (LF) character.

The closing ?> tag MUST be omitted from files containing only PHP.

148.4 Lines

There MUST NOT be a hard limit on line length.

The soft limit on line length MUST be 120 characters; automated style checkers MUST warn but MUST NOT error at the soft limit.

Lines SHOULD NOT be longer than 80 characters; lines longer than that SHOULD be split into multiple subsequent lines of no more than 80 characters each.

There MUST NOT be trailing whitespace at the end of lines.

Blank lines MAY be added to improve readability and to indicate related blocks of code except where explictly forbidden.

There MUST NOT be more than one statement per line.

148.5 Intenting

Code MUST use an indent of 4 spaces for each indent level, and MUST NOT use tabs for indenting.

148.6 Keywords

PHP keywords MUST be in lower case.

The PHP types and keywords array, int, true, object, float, false, mixed, bool, null, numeric, string, void and resource MUST be in lower case.

Short form of type keywords MUST be used in both code and documentation blocks i.e. bool instead of boolean, int instead of integer etc.

148.7 Declaration

The header of a PHP file may consist of a number of different blocks. If present, each of the blocks below MUST be separated by a single blank line, and MUST NOT contain a blank line. Each block MUST be in the order listed below, although blocks that are not relevant may be omitted.

- File-level docblock.
- One or more declare statements.
- The namespace declaration of the file.
- One or more class-based use statements.
- One or more function-based use statements.
- One or more constant-based use statements.

- The remainder of the code in the file.

When a file contains a mix of HTML and PHP, any of the above sections may still be used. If so, they MUST be present at the top of the file, even if the remainder of the code consists a closing PHP tag and then a mixture of HTML and PHP.

When the opening <?php tag is on the first line of the file, it MUST be on its own line with no other statements unless it is a file containing markup outside of PHP opening and closing tags.

The following example illustrates a complete list of all blocks:

```
<?php

/**
 * This file contains an example of coding styles.
 */

declare(strict_types=1);

namespace Vendor\Package;

use Vendor\Package\{ClassA as A, ClassB, ClassC as C};
use Vendor\Package\Namespace\ClassD as D;
use Vendor\Package\AnotherNamespace\ClassE as E;

use function Vendor\Package\{functionA, functionB, functionC};
use function Another\Vendor\function D;

use const Vendor\Package\{CONSTANT_A, CONSTANT_B, CONSTANT_C};
use const Another\Vendor\CONSTANT_D;

/**
 * FooBar is an example class.
 */
class FooBar
{
    // ... additional PHP code ...
}
```

Compound namespaces with a depth of more than two MUST not be used. Therefore the following is the maximum compounding depth allowed:

```
<?php

use Vendor\Package\Namespace\{
```

```
    SubnamespaceOne\ClassA,
    SubnamespaceOne\ClassB,
    SubnamespaceTwo\ClassY,
    ClassZ,
};
```

And the following would not be allowed:

```
<?php

use Vendor\Package\Namespace\{
    SubnamespaceOne\AnotherNamespace\ClassA,
    SubnamespaceOne\ClassB,
    ClassZ,
};
```

When wishing to declare strict types in files containing markup outside PHP opening and closing tags MUST, on the first line, include an opening php tag, the strict types declaration and closing tag.

```
<?php declare(strict_types=1); ?>
<html>
<body>
    <?php
        // ... additional PHP code ...
    ?>
</body>
</html>
```

Declare statements MUST contain no spaces and MUST look like declare(strict_types=1);.

Block declare statements are allowed and MUST be formatted as below. Note position of braces and spacing:

```
declare(ticks=1) {
    //some code
}
```

148.8 Classes

The term ”class” refers to all classes, interfaces, and traits.

Any closing brace must not be followed by any comment or statement on the same line.

When instantiating a new class, parenthesis MUST always be present even when there are no arguments passed to the constructor.

```
new Foo();
```

148.9 Inherit

The extends and implements keywords MUST be declared on the same line as the class name.

The opening brace for the class MUST go on its own line; the closing brace for the class MUST go on the next line after the body.

Opening braces MUST be on their own line and MUST NOT be preceded or followed by a blank line.

Closing braces MUST be on their own line and MUST NOT be preceded by a blank line.

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements \ArrayAccess, \Countable
{
    // constants, properties, methods
}
```

Lists of implements and extends MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one interface per line.

```
<?php
namespace Vendor\Package;

use FooClass;
use BarClass as Bar;
use OtherVendor\OtherPackage\BazClass;

class ClassName extends ParentClass implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // constants, properties, methods
}
```

148.10 Traits

The use keyword used inside the classes to implement traits MUST be declared on the next line after the opening brace.

```
<?php
namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

Each individual Trait that is imported into a class MUST be included one-per-line, and each inclusion MUST have its own use statement.

```
<?php
namespace Vendor\Package;

use Vendor\Package\FirstTrait;
use Vendor\Package\SecondTrait;
use Vendor\Package\ThirdTrait;

class ClassName
{
    use FirstTrait;
    use SecondTrait;
    use ThirdTrait;
}
```

When the class has nothing after the use declaration, the class closing brace MUST be on the next line after the use declaration.

```
<?php
namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;
}
```

Otherwise it MUST have a blank line after the use declaration.

```
<?php
namespace Vendor\Package;

use Vendor\Package\FirstTrait;

class ClassName
{
    use FirstTrait;

    private $property;
}
```

148.11 Properties

Visibility MUST be declared on all properties.

The var keyword MUST NOT be used to declare a property.

There MUST NOT be more than one property declared per statement.

Property names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

A property declaration looks like the following.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public $foo = null;
}
```

148.12 Methods

Visibility MUST be declared on all methods.

Method names MUST NOT be prefixed with a single underscore to indicate protected or private visibility. That is, an underscore prefix explicitly has no meaning.

Method and function names MUST NOT be declared with a space after the method name. The opening brace MUST go on its own line, and the closing brace MUST go on the next line following

the body. There MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis.

A method declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function fooBarBaz($arg1, &$arg2, $arg3 = [])
    {
        // method body
    }
}
```

A function declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php

function fooBarBaz($arg1, &$arg2, $arg3 = [])
{
    // function body
}
```

148.13 Arguments

In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Method and function arguments with default values MUST go at the end of the argument list.

Method and function argument scalar type hints MUST be lowercase.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function foo(int $arg1, &$arg2, $arg3 = [])
    {
        // method body
```

```
    }
}
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line.

When the argument list is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

```
<?php
namespace Vendor\Package;

class ClassName
{
    public function aVeryLongMethodName(
        ClassTypeHint $arg1,
        &$arg2,
        array $arg3 = []
    ) {
        // method body
    }
}
```

When you have a return type declaration present there MUST be one space after the colon with followed by the type declaration. The colon and declaration MUST be on the same line as the argument list closing parentheses with no spaces between the two characters. The declaration keyword (e.g. string) MUST be lowercase.

```
<?php
declare(strict_types=1);

namespace Vendor\Package;

class ReturnTypeVariations
{
    public function functionName($arg1, $arg2): string
    {
        return 'foo';
    }
}
```

148.14 Visibility

When present, the abstract and final declarations MUST precede the visibility declaration.

When present, the static declaration MUST come after the visibility declaration.

```
<?php
namespace Vendor\Package;

abstract class ClassName
{
    protected static $foo;

    abstract protected function zim();

    final public static function bar()
    {
        // method body
    }
}
```

148.15 Calling

When making a method or function call, there MUST NOT be a space between the method or function name and the opening parenthesis, there MUST NOT be a space after the opening parenthesis, and there MUST NOT be a space before the closing parenthesis. In the argument list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

```
<?php

bar();
$foo->bar($arg1);
Foo::bar($arg2, $arg3);
```

Argument lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument per line. A single argument being split across multiple lines (as might be the case with an anonymous function or array) does not constitute splitting the argument list itself.

```
<?php

$foo->bar(
    $longArgument,
```

```
    $longerArgument,
    $muchLongerArgument
);
```

```
<?php

somefunction($foo, $bar, [
  // ...
], $baz);

$app->get('/hello/{name}', function ($name) use ($app) {
    return 'Hello ' . $app->escape($name);
});
```

148.16 Structure

The general style rules for control structures are as follows:

- There MUST be one space after the control structure keyword
- There MUST NOT be a space after the opening parenthesis
- There MUST NOT be a space before the closing parenthesis
- There MUST be one space between the closing parenthesis and the opening brace
- The structure body MUST be indented once
- The closing brace MUST be on the next line after the body

The body of each structure MUST be enclosed by braces. This standardizes how the structures look, and reduces the likelihood of introducing errors as new lines get added to the body.

148.16.1 if/elseif/else

An if structure looks like the following. Note the placement of parentheses, spaces, and braces; and that else and elseif are on the same line as the closing brace from the earlier body.

```
<?php

if ($expr1) {
    // if body
} elseif ($expr2) {
    // elseif body
} else {
    // else body;
}
```

The keyword elseif SHOULD be used instead of else if so that all control keywords look like single words.

148.16.2 switch/case

A switch structure looks like the following. Note the placement of parentheses, spaces, and braces. The case statement MUST be indented once from switch, and the break keyword (or other terminating keyword) MUST be indented at the same level as the case body. There MUST be a comment such as // no break when fall-through is intentional in a non-empty case body.

```
<?php

switch ($expr) {
    case 0:
        echo 'First case, with a break';
        break;
    case 1:
        echo 'Second case, which falls through';
        // no break
    case 2:
    case 3:
    case 4:
        echo 'Third case, return instead of break';
        return;
    default:
        echo 'Default case';
        break;
}
```

148.16.3 while/do-while

A while statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

while ($expr) {
    // structure body
}
```

Similarly, a do while statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

do {
    // structure body;
} while ($expr);
```

148.16.4 for

A for statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

for ($i = 0; $i < 10; $i++) {
    // for body
}
```

148.16.5 foreach

A foreach statement looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

foreach ($iterable as $key => $value) {
    // foreach body
}
```

148.16.6 try/catch/finally

A try-catch-finally block looks like the following. Note the placement of parentheses, spaces, and braces.

```
<?php

try {
    // try body
} catch (FirstThrowableType $e) {
    // catch body
} catch (OtherThrowableType $e) {
    // catch body
} finally {
    // finally body
```

```
}
```

148.17 Operators

All binary and ternary (but not unary) operators MUST be preceded and followed by at least one space. This includes all arithmetic, comparison, assignment, bitwise, logical (excluding ! which is unary), string concatenation, and type operators.

Other operators are left undefined.

```
<?php

if ($a === $b) {
    $foo = $bar ?? $a ?? $b;
} elseif ($a > $b) {
    $variable = $foo ? 'foo' : 'bar';
}
```

148.18 Closures

Closures MUST be declared with a space after the function keyword, and a space before and after the use keyword.

The opening brace MUST go on the same line, and the closing brace MUST go on the next line following the body.

There MUST NOT be a space after the opening parenthesis of the argument list or variable list, and there MUST NOT be a space before the closing parenthesis of the argument list or variable list.

In the argument list and variable list, there MUST NOT be a space before each comma, and there MUST be one space after each comma.

Closure arguments with default values MUST go at the end of the argument list.

A closure declaration looks like the following. Note the placement of parentheses, commas, spaces, and braces:

```
<?php

$closureWithArgs = function ($arg1, $arg2) {
    // body
};

$closureWithArgsAndVars = function ($arg1, $arg2) use ($var1, $var2) {
```

```
    // body
};
```

Argument lists and variable lists MAY be split across multiple lines, where each subsequent line is indented once. When doing so, the first item in the list MUST be on the next line, and there MUST be only one argument or variable per line.

When the ending list (whether of arguments or variables) is split across multiple lines, the closing parenthesis and opening brace MUST be placed together on their own line with one space between them.

The following are examples of closures with and without argument lists and variable lists split across multiple lines.

```
<?php

$longArgs_noVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) {
    // body
};

$noArgs_longVars = function () use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};

$longArgs_longVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
    // body
};
```

```
$longArgs_shortVars = function (
    $longArgument,
    $longerArgument,
    $muchLongerArgument
) use ($var1) {
   // body
};

$shortArgs_longVars = function ($arg) use (
    $longVar1,
    $longerVar2,
    $muchLongerVar3
) {
   // body
};
```

Note that the formatting rules also apply when the closure is used directly in a function or method call as an argument.

```
<?php

$foo->bar(
    $arg1,
    function ($arg2) use ($var1) {
        // body
    },
    $arg3
);
```

148.19 Anonymous

Anonymous Classes MUST follow the same guidelines and principles as closures in the above section.

```
<?php

$instance = new class {};
```

The opening parenthesis MAY be on the same line as the class keyword so long as the list of implements interfaces does not wrap. If the list of interfaces wraps, the parenthesis MUST be placed on the line immediately following the last interface.

```php
<?php

// Parenthesis on the same line
$instance = new class extends \Foo implements \HandleableInterface {
    // Class content
};

// Parenthesis on the next line
$instance = new class extends \Foo implements
    \ArrayAccess,
    \Countable,
    \Serializable
{
    // Class content
};
```

Chapter 149

PSR-13

Part XXXI

Laravel

Chapter 150

Overview

150.1 Framework

框架可以避免重新造轮子来构建 Web 应用，而且框架通常都抽象了许多底层常用的逻辑，并提供了简便的方法来完成常见的任务。

虽然并不一定要在每个项目中都使用框架，因为有时候原生的 PHP 才是正确的选择，但是如果需要一个框架，那么有如下三种主要类型：

- 微型框架
- 全栈框架
- 组件框架

微型框架基本上是一个封装的路由，用来转发 HTTP 请求至一个闭包、控制器或方法等，尽可能地加快开发的速度，有时还会使用一些类库来帮助开发（例如基本的数据库封装等），因此微型框架适合用来构建 HTTP 的服务。

在微型框架的功能齐备后往往演化为全栈框架，这些全栈框架通常会提供 ORMs 和身份认证扩展包等。

组件框架是多个独立的类库所结合起来的，而且不同的组件框架可以一起使用在微型或是全栈框架上。

实际上，「组件」是另一种建立、发布及推动开源的方式，现在存在的组件库主要包括 Packagist 和 PEAR，而且它们都有用来安装和升级的命令行工具。

此外，还有基于组件的构成的框架的提供商提供不包含框架的组件，这些项目通常和其他的组件或者特定的框架没有依赖关系。例如，可以使用 FuelPHP 验证类库而不使用 FuelPHP 整个框架，或者使用 Illuminate 组件来实现 Laravel 框架解耦。

150.2 Namespace

Laravel 框架把已有的技术（例如架构思想、设计模式和 PHP 新语法）组合到了一定的告诉，其中 PHP 新语法解决了一些棘手的问题。

PHP 的命名空间、匿名函数、反射机制和后期静态绑定等新语法使得 Laravel 框架简洁而且容易扩展。

命名空间最初是一种被设计用来解决命名冲突的包装类、函数和常量的方法，同时命名空间为组件化开发提供了的可能。

命名空间组织文件时使用某个组件的文件的路径关联到命名空间，最终可以直接通过命名空间找到相应的文件，这样借助 Composer 就可以方便地管理组件包的文件关系，同时依据 PSR 规范就可以把不同的组件包组合到一起。

如果需要使用命名空间来封装代码，那么命名空间需要使用 namespace 关键字声明并放在文件开头，例如：

```
<?php
namespace App\Http;

use Illuminate\Foundation\Http\Kernel as HttpKernel;
class Kernel extends HttpKernel {
   // class definition
}
```

上述的示例定义了命名空间 App\Http，并且在命名空间下定义 Kernel 类，因此 Kernel 类的完整名称就是 App\Http\Kernel。

从本质上来说，Kernel 类文件的文件路径和命名空间是独立的，在任何目录下都可以定义 App\Http 命名空间。

PSR 命名规范提出为了后期文件包含的方便可以把命名空间和文件路径定义为相同的名称，文件名和类名定义为相同的名称，这样通过一个类的完整名称就可以确定这个类所在文件相对于根目录的位置，同时也方便了后期文件包含。

PHP 支持使用 __NAMESPACE__ 魔术常量和 namespace 关键字来获取和使用当前命名空间。

__NAMESPACE__ 魔术常量可以直接获取当前命名空间名称的字符串，namespace 关键字可以显式访问当前命名空间。

全局代码（不包含任何命名空间）的命名空间是一个空字符串，没有定义命名空间就是全局空间，相当于根空间（使用“/”表示根空间）。

```
<?php
namespace App\Http;
echo __NAMESPACE__."\n";
class Kernel {}
```

```
$a = new namespace\Kernel();
var_dump($a);
```

上述示例的输出如下：

```
App\Http
class App\Http\Kernel#1 (0) {
}
```

用户可以使用 use 关键字并通过相对路径、限定的相对路径和绝对路径三种方式来使用命名空间

- 非限定名称或不包含前缀的类名称。例如，`$a=new Kernel();`
  1. 如果当前命名空间为 App\Http，那么 Kernel 被解析为 App\Http\Kernel。

     ```
     class App\Http\Kernel#1 (0) {
     }
     ```

  2. 如果当前命名空间为根空间（全局的），那么 Kernel 被解析为\Kernel。

     ```
     class Kernel#1 (0) {
     }
     ```

- 限定名称或包含前缀的类名称。例如，`$a=new App\Http\Kernel();`
  1. 如果当前命名空间为\Illiminate，那么 Kernel 被解析为\Illiminate\App\Http\Kernel。

     ```
     class Illiminate\App\Http\Kernel#1 (0) {
     }
     ```

  2. 如果当前命名空间为根空间（全局的），那么 Kernel 被解析为\App\Http\Kernel。

     ```
     class App\Http\Kernel#1 (0) {
     }
     ```

- 完全限定名称或包含全局前缀操作符(“\”)的名称。例如，`$a=new \App\Http\Kernel();`

  ```
  class App\Http\Kernel#1 (0) {
  }
  ```

如果命名空间的名字过长，PHP 还允许通过导入外部引用或别名引用的方式来指定相应的类和命名空间，即可以为类名称使用别名，也可以为命名空间使用别名。

```
namespace App\Http\Controller\Auth;

use App\Http\Controller\Controller;
use App\Model\Category as DataCategory;
use App\Model;

class AuthController extends Controller{
```

```
}
//实例化App\Http\Controller\Controller对象
$obj1=new Controller();

//实例化App\Http\Controller\Auth\AuthController对象
$obj2=new AuthController();

//实例化App\Model\Category对象
$obj3=new DataCategory();

//实例化App\Model\Records对象
$obj4=new Model\Records();

//实例化Model\Records对象
$obj5=new \Model\Records();
```

PHP 的命名空间只支持导入类，不支持导入函数或常量，而且对于命名空间中的类来说，其最前面是不允许有反斜杠的。

导入的名称都是完全限定的，不会根据命名空间进行相对解析，只有在实例化一个对象时可以通过完全限定名称来指定一个类，而且也不会进行相对解析，仅仅针对最前面没有反斜杠的实例化类名进行相对解析。

- 完全限定名称的类、函数和常量可以直接直接解析。

```
<?php
$a = new \A\B();
```

- 所有非限定名称和非完全限定名称的类、函数和常量根据当前导入的命名空间进行转换。

```
<?php
use A\B\C;
$a = new C\D\E();// 等同于$a = A\B\C\D\E();
```

- 命名空间内部的所有的没有根据导入规则转换的非限定名称和非完全限定名称都会其前面加上当前的命名空间名称。

```
<?php
namespace A\B;

$a = new C\D\E();// 等同于$a = new A\B\C\D\E();
```

- 命名空间内部对非限定名称和非完全限定名称的函数进行调用时，首先在当前命名空间下解析，入股查找不到就在全局空间下查找。

```php
<?php
namespace A\B;

$a = new A\B();
$a->foo();
```

1. 在当前命名空间中查找 A\B\foo() 函数;
2. 尝试查找并调用全局空间（“\”）中的 foo() 函数。

- 命名空间内部对非限定名称和非完全限定名称的类进行调用时，只会在当前命名空间下解析。

```php
<?php
namespace A\B;

$a = new C();// 等同于$a = new A\B\C();
$b = new \C();// 实例化全局命名空间中的全局类C。
```

150.3 Autoloading

include 和 require 关键字都用于包含并运行指定文件，只是处理失败的方式不同。

- require 在出错时产生 E_COMPILE_ERROR 级别的错误并导致脚本运行中止。
- include 在出错时产生 E_WARNING 级别的错误，仅发出警告并不会中止脚本运行。

在包含文件时如果只包含一个文件名，那么 PHP 首先在 php.ini 的 include_path 指定的目录下查找文件，如果没有找到就会到脚本文件所在的工作目录下查找文件，最后如果找不到就产生上述的错误之一。

默认情况下，include_path 的配置如下:

```
include_path=".;/usr/share/php"
```

如果在包含文件时指定的文件名包含路径，则系统就只根据其中指定的相对路径或绝对路径去查找文件。例如，如果文件名以“../”开头，那么 PHP 只在当前目录的父目录下查找文件。

如果包含的文件被找到，那么包含文件继承被包含文件定义的变量，并且从包含文件的地方开始，被包含文件可用的任何变量在包含的文件中都可用，同时在被包含文件中定义的类、函数或常量都具有全局作用域。

除了使用 include 和 require 以手动方式来包含文件之外，PHP 还支持对类进行自动加载。

类的自动加载可以通过魔术方法 __autoload($class) 实现，也可以通过 spl_autoload_register 注册一个自动加载方法。

150.3.1 __autoload()

```
<?php
function __autoload($class) {
   require_once($class.".php'');
}
```

如果在使用一个类时，PHP 发现这个类没有被当前文件包含，就会自动调用 __autoload($class) 魔术方法（其中的 $class 就是要使用的类名）。

150.3.2 spl_autoload_register()

__autoload() 只能定义一次，因此用户实际应用中往往使用 spl_autoload_register() 注册自定义的函数作为自动加载类的实现。

spl_autoload_register() 允许把多个类自动加载函数注册到队列中，也就是说它创建了 __autoload() 函数的队列，并且在调用时按照定义时的顺序逐个执行。

```
bool spl_autoload_register ([ callable $autoload_function [, bool $throw = true [, bool
    $prepend = false ]]] )
```

向 __autoload 队列注册给定的函数。如果该队列中的函数尚未激活，则激活它们。

对于代码中已经实现了 __autoload() 函数，必须显式注册到 __autoload 队列中，否则 spl_autoload_register() 将使用 spl_autoload() 或 spl_autoload_call() 取代 __autoload() 函数。

表 150.1: spl_autoload_register() 参数

参数名	说明
autoload_function	要注册的自动加载函数，没有提供时将自动注册 spl_autoload() 作为默认实现
throw	自动加载函数无法注册时，spl_autoload_register() 是否抛出异常
prepend	是否把自动加载函数注册到自动加载函数队列的头部（默认注册到队列的尾部）

Example 151 使用 spl_autoload_register() 代替 __autoload()

```
<?php
// function __autoload($class) {
//    include 'classes/'.$class.'.class.php';
// }
function my_autoloader($class) {
```

```
    include 'classes/'.$class.'.class.php';
}
spl_autoload_register('my_autoloader');
```

PHP 支持使用匿名函数来实现注册自动加载函数，例如：

```
<?php
spl_autoload_register(function($class){
    include 'classes/'.$class.'.class.php';
});
```

Example 152 使用 spl_autoload_register() 注册未加载的类

```
<?php
namespace Foobar;

class Foo {
    static public function test($name) {
        print '[['.$name.']]';
    }
}
spl_autoload_register(__NAMESPACE__.'\Foo::test');

new InexistentClass;
```

上述示例的输出如下：

```
[[Foobar\InexistentClass]]
Fatal error: Class 'Foobar\InexistentClass' not found in ...
```

spl_autoload_register() 函数加载的自动加载函数可以是全局函数，也可以是某个类实例对象的函数，即通过 `array(" 对象名"," 函数名")` 的方式进行注册。

```
<?php
class ClassLoader{
    public function register($prepend = false){
        spl_autoload_register(array($this,'loadClass'),true,$prepend);
    }
}
```

Laravel 框架通过 spl_autoload_register() 实现类自动加载函数的注册时，类的自动加载函数队列中包含了两个类的自动加载函数，分别是 Composer 生成的基于 PSR 规范的自动加载函数和 Laravel 框架核心别名的自动加载函数。

在 public/index.php 中包含了 Composer 生成的自动加载函数，例如：

```
<?php
require __DIR__.'/../bootstrap/autoload.php';

$app = require_once __DIR__.'/../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

$response = $kernel->handle($request=Illuminate\Http\Request::capture());
$response->send();
$kernel->terminate($request,$response);
```

在 bootstrap/autoload.php 中包含了 Composer 自动加载器，例如：

```
<?php
define('LARAVEL_START',microtime(true));

require __DIR__.'/../vendor/autoload.php';

$compiledPath = __DIR__.'/cache/compiled.php';
if(file_exists($compiledPath)){
   require $compiledPath;
}
```

public/index.php 是 Laravel 框架的请求入口文件，因此首先会包含 bootstrap 目录下的自动加载文件——autoload.php。

bootstrap/autoload.php 继续加载 vendor 目录下的自动加载文件，并且在依赖文件加载完毕后进入 HTTP 请求捕获、处理和响应，最终清理 HTTP 请求数据。

150.4 Magic Method

PHP 提供的魔术方法和魔术常量常用来提供对 PHP 运行环境和过程的控制和检测。

和普通方法根据用户的实现方式进行调用的方式不同，魔术方法通常并不会由用户主动调用，而是在特定的时机并 PHP 自动调用。

魔术方法可以通俗地理解为系统监听方法，在事件发生时才触发执行，类似于嵌入式系统中的中断函数。

用户不能定义以“__”开头的方法，只有魔术方法可以以“__”开头。例如，__construct() 方法就是在创建实例完成后自动调用的一个魔术方法。

表 150.2: 魔术方法

方法名	说明
__construct()	在每次实例化对象时先调用该方法，可以在调用对象之前进行对象初始化
__destruct()	在对象的所有引用都被删除或对象被显式销毁时执行
__set()	在给不可访问的属性赋值时，__set() 方法被调用来进行属性重载
__get()	在读取不可访问的属性时调用
__isset()	对不可访问的属性调用 isset() 或 empty() 时调用 __isset()
__unset()	对不可访问的属性调用 unset() 时调用 __unset()
__sleep()	serialize() 方法检查类中是否有 __sleep()。如果 __sleep() 方法存在，那么该方法就会在任何序列化操作之前运行，可以清除对象并返回一个包含有该对象中应该被序列化的所有变量名的数组。
__wakeup()	unserialize() 方法检查类中是否有 __wakeup()。如果 __wakeup() 方法存在，那么该方法可以被用来执行对象重建。
__toString()	用于一个类被当做字符串时的输出操作。例如，执行 `echo $obj;` 就会调用 __toString() 方法。
__invoke()	尝试以调用函数的方式来调用一个对象时就会自动调用 __invoke() 方法。例如，执行 `$obj();` 就会调用 __invoke() 方法。
__clone()	在新创建对象（复制生成对象）时会尝试调用 __clone() 方法（如果定义了 __clone() 方法），而且可以用来修改属性的值。
__call()	在对象中调用一个不可访问的方法时会调用 __call() 方法。
__callStatic()	在静态方法中调用一个不可访问的方法时会调用 __callStatic() 方法。

方法名	说明
__autoload()	在试图使用尚未被定义的类时自动调用 __autoload() 方法。

Laravel 框架中的 Illuminate\Database\Eloquent\Model 类使用魔术方法来实现对 ORM 对象的属性和方法的操作。

```
<?php
namespace Illuminate\Database\Eloquent;

abstract class Model implements ArrayAccess, Arrayable, Jsonable, JsonSerializable,
    QueueableEntity, UrlRoutable
{
    /**
     * 实例化一个新的Eloquent模型实例
     */
    public function __construct(array $attributes = [])
    {
        $this->bootIfNotBooted();
        $this->syncOriginal();
        $this->fill($attributes);
    }

    /**
     * 在模型实例中索引不存在的属性
     */
    public function __get($key)
    {
        return $this->getAttribute($key);
    }

    /**
     * 在模型实例中设置不存在的属性
     */
    public function __set($key, $value)
    {
        $this->setAttribute($key, $value);
    }

    /**
```

```
 * 判断模型实例中是否存在该属性
 */
public function __isset($key)
{
    return ! is_null($this->getAttribute($key));
}

/**
 * 注销模型实例中的一个属性
 */
public function __unset($key)
{
    unset($this->attributes[$key], $this->relations[$key]);
}

/**
 * 处理模型实例中不存在的函数调用
 */
public function __call($method, $parameters)
{
    if (in_array($method, ['increment', 'decrement'])) {
        return call_user_func_array([$this, $method], $parameters);
    }

    $query = $this->newQuery();

    return call_user_func_array([$query, $method], $parameters);
}

/**
 * 处理模型示例中不存在的静态函数调用
 */
public static function __callStatic($method, $parameters)
{
    $instance = new static;

    return call_user_func_array([$instance, $method], $parameters);
}

/**
 * 把模型实例转换为JSON字符串
```

```
     */
    public function __toString()
    {
        return $this->toJson();
    }

    /**
     * 反序列化模型时判断是否需要启动
     */
    public function __wakeup()
    {
        $this->bootIfNotBooted();
    }
}
```

修改 Laravel 框架的入口文件来测试魔术方法的使用时，可以引入 Model 类并扩展出一个 User 类，然后实例化 User 类就可以触发相应的魔术方法调用。

```
<?php
require __DIR__.'/../bootstrap/autoload.php';
$app = require_once __DIR__.'/../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
   protected $table = 'users';
}

$user = new User();//__construct()初始化对象
$user->name = 'test';//__set($key,$value)设置属性
echo $user->name;//__get($key)获取属性
isset($user->name);//__isset($key)判断属性是否存在
unset($user->name);//__unset($key)删除属性
$user->find(1);//__call($method,$parameters)调用方法
User::find(1);//__callStatic($method,$parameters)调用静态方法
echo $user;//调用__toString()方法
$us = serialize($user);//调用__sleep()
$us1 = unserialize($us);//调用__wakeup()方法
```

150.5 Magic Constant

PHP 为其脚本提供了一些预定义常量，而且不同的扩展库本身也提供了一些常量（只有加载了对应的扩展库才会生效）。

PHP 扩展可以使用 dl() 函数动态加载，也可以编译进 PHP 核心成为内置模块。

PHP 为运行环境提供了 8 个魔术常量，而且这些魔术常量的值会随着代码中的位置而改变。例如，__LINE__ 依赖于其在 PHP 脚本中所处的行来决定。

表 150.3: 魔术常量

常量名	说明
__LINE__	当前行号
__FILE__	当前文件的完整路径和文件名。如果被用在被包含文件中，则返回被包含的文件名。
__DIR__	当前文件所在目录。如果被用在被包含文件中，则返回被包含的文件所在的目录，等价于 `dirname(__FILE__);`，而且除非是根目录，否则目录名中不包括末尾的斜杠。
__FUNCTION__	当前所处的函数名并区分大小写。
__CLASS__	当前所处的类名或 Trait 名并区分大小写 (包括其被声明的命名空间)，如果用在 Trait 方法中，返回调用 Trait 方法的类名。
__TRAIT__	当前所处的 Trait 名并区分大小写 (包括其被声明的命名空间)。
__METHOD__	当前所处的方法名（并区分大小写)。
__NAMESPACE__	当前命名空间并区分大小写。

Laravel 框架使用魔术常量来确定文件所在的目录，这样就可以通过严格对应的目录结构找到其他文件的位置实现类的自动加载等。

在 Laravel 框架的入口文件 public/index.php 中根据 __DIR__ 获取当前文件的目录，接下来再根据目录的相对位置启动其他文件代码。

```
<?php
require __DIR__.'/../bootstrap/autoload.php';

$app = require_once __DIR__.'/../bootstrap/app.php';
$kernel = $app->make(Illuminate\Contracts\Http\Kernel::class);
$response = $kernel->handle($request=Illuminate\Http\Request::capture());
```

```
$response->send();
$kernel->terminate($request,$response);
```

150.6 Late Static Binding

后期静态绑定（Late Static Binding）可以用于在继承范围内引用静态调用的类，也就是在类的继承过程中使用的类不再是当前类，而是调用的类。

后期静态绑定使用 static 关键字来实现，后期静态绑定中的“static::”不再被解析为定义当前方法所在的类，而是在实际运行时计算得到的（也就是在运行时最初调用的类）。

后期静态绑定不只在静态方法的调用中，同样适用于非静态方法的调用，而且调用的方式和静态方法相同，都是在调用时动态确定的。

```
<?php
class A {
   public static function call() {
      echo 'class A';
   }
   public static function test() {
      self::call();
      static::call();
   }
}

class B extends A {
   public static function call() {
      echo 'class B';
   }
}
B::test()
//输出结果:
// class A
// class B
```

在调用 test() 方法时，self:: 直接调用本类中的方法，static:: 则可以根据调用 test() 方法的类来决定 static:: 的值。

static 的值只有在调用时才能确定下来，self 则是在定义时就确定下来。

```
class A {
   public function call() {
      echo 'instance from A';
   }
```

```
  public function test() {
     self::call();
     static::call();
  }
}
class B extends A {
  public function call() {
    echo 'instance from B';
  }
}
$b = new B();
$b->test();
// 输出结果:
// instance from A
// instance from B
```

除了可以用于静态方法和普通方法，后期静态绑定还可以用于对象实例化中。

- static 根据运行时调用的类来决定实例化对象；
- self 根据所在位置的类来决定实例化对象。

```
<?php
class A {
  public static function create() {
     $self = new self();
     $static = new static();
     return array($self,$static);
  }
}
class B extends A {

}
$arr = B::create();
foreach ($arr as $value) {
  var_dump($value);
}
// 输出结果如下:
// object(A) [1]
// object(B) [2]
```

Laravel 框架大量使用了静态方法、普通方法和对象实例化的后期静态绑定。例如，Illuminate\Database\Eloquent\Model 类中的 `$model = new static($attributes);` 和 `return static::($attributes);`。

Illuminate\Database\Eloquent\Model 类是一个抽象类，因此其实现类在调用这些方法时，最终动态绑定的都是这个类的实现类（而非 Model 抽象类）。

```
// 保存模型类实例中的数据并返回该模型类实例
public static function create(array $attributes = [])
{
    $model = new static($attributes);
    $model->save();
    return $model;
}
// 获取第一个符合条件的数据或创建一个新的
public function firstOrCreate(array $attributes)
{
   if(!is_null($instance = static::where($attributes)->first())){
     return $instance;
   }
   return static::create($attributes);
}
```

Chapter 151

Composer

Laravel 框架下的 vendor 目录本身是 Composer 生成的依赖包目录，其内部的自动加载文件也是 Composer 文件生成的，用来自动加载依赖包中的所有文件。

151.1 ClassLoader

Composer 在创建依赖管理时会在 vendor 目录下创建一个 autoload.php 和一个 composer 目录，其中 composer 目录包含了类自动加载函数注册的相关实现，这个 autoload.php 就是其对外提供的接口，通过包含该文件来完成类自动加载函数的注册。

实际上，vendor/autoload.php 会进一步包含 vendor/composer 目录中的 autoload_real.php 文件。

```
require_once __DIR__ . '/composer/autoload_real.php';

return ComposerAutoloaderInitca049c87f29a6ff5956132f96c58718a::getLoader();
```

vendor/composer/autoload_real.php 定义了一个类，这个类中定义了 getLoader() 函数。

```
class ComposerAutoloaderInitca049c87f29a6ff5956132f96c58718a
{
    private static $loader;

    public static function loadClassLoader($class)
    {
        if ('Composer\Autoload\ClassLoader' === $class) {
            require __DIR__ . '/ClassLoader.php';
        }
    }
```

```
public static function getLoader()
{
    if (null !== self::$loader) {
        return self::$loader;
    }

    spl_autoload_register(array('
        ComposerAutoloaderInitca049c87f29a6ff5956132f96c58718a', 'loadClassLoader'),
        true, true);
    self::$loader = $loader = new \Composer\Autoload\ClassLoader();
    spl_autoload_unregister(array('
        ComposerAutoloaderInitca049c87f29a6ff5956132f96c58718a', 'loadClassLoader'));

    $useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HHVM_VERSION');
    if ($useStaticLoader) {
        require_once __DIR__ . '/autoload_static.php';

        call_user_func(\Composer\Autoload\
            ComposerStaticInitca049c87f29a6ff5956132f96c58718a::getInitializer($loader
            ));
    } else {
        $map = require __DIR__ . '/autoload_namespaces.php';
        foreach ($map as $namespace => $path) {
            $loader->set($namespace, $path);
        }

        $map = require __DIR__ . '/autoload_psr4.php';
        foreach ($map as $namespace => $path) {
            $loader->setPsr4($namespace, $path);
        }

        $classMap = require __DIR__ . '/autoload_classmap.php';
        if ($classMap) {
            $loader->addClassMap($classMap);
        }
    }

    $loader->register(true);

    if ($useStaticLoader) {
        $includeFiles = Composer\Autoload\
```

```
            ComposerStaticInitca049c87f29a6ff5956132f96c58718a::$files;
        } else {
            $includeFiles = require __DIR__ . '/autoload_files.php';
        }
        foreach ($includeFiles as $fileIdentifier => $file) {
            composerRequireca049c87f29a6ff5956132f96c58718a($fileIdentifier, $file);
        }

        return $loader;
    }
}

function composerRequireca049c87f29a6ff5956132f96c58718a($fileIdentifier, $file)
{
    if (empty($GLOBALS['__composer_autoload_files'][$fileIdentifier])) {
        require $file;

        $GLOBALS['__composer_autoload_files'][$fileIdentifier] = true;
    }
}
```

getLoader() 函数首先实例化 Composer\Autoload\ClassLoader 类，然后通过这个类实例添加相关的文件路径配置，包括命名空间的配置、PSR-4 规范配置、类映射配置。

- 命名空间配置：autoload_namespace.php
- PSR-4 规范配置：autoload_psr4.php
- 类映射配置：autoload_classmap.php

```
<?php
/...
self::$loader = $loader = new \Composer\Autoload\ClassLoader();
//...
$useStaticLoader = PHP_VERSION_ID >= 50600 && !defined('HHVM_VERSION');
if ($useStaticLoader) {
    require_once __DIR__ . '/autoload_static.php';

    call_user_func(\Composer\Autoload\ComposerStaticInitca049c87f29a6ff5956132f96c58718a
        ::getInitializer($loader));
} else {
    $map = require __DIR__ . '/autoload_namespaces.php';
    foreach ($map as $namespace => $path) {
        $loader->set($namespace, $path);
    }
```

```
    $map = require __DIR__ . '/autoload_psr4.php';
    foreach ($map as $namespace => $path) {
        $loader->setPsr4($namespace, $path);
    }

    $classMap = require __DIR__ . '/autoload_classmap.php';
    if ($classMap) {
        $loader->addClassMap($classMap);
    }
}

$loader->register(true);
```

`$loader->register(true)` 执行注册类自动加载函数到队列，最后则加载全局文件（即 autoload_files.php 配置的内容）。

类的自动加载函数在 Composer\Autoload\ClassLoader 类中实现，实例化 ClassLoader 类并将类的命名空间与文件路径的对应关系注册到相应属性中，然后再通过实例方法 `register($prepend=fals` 来注册一个类自动加载函数，也就是 ClassLoader 类实例的 loadClass() 方法，并且将其注册在类自动加载函数队列的末尾。

当使用一个未包含的类名时，就会自动调用 loadClass() 方法并通过参数获取包含命名空间的类名信息，接着根据类的命名空间与文件路径的对应关系查找文件路径，最后通过 includeFile() 函数包含该文件，这样就实现了类的自动加载。

```
<?php
public function loadClass($class)
{
    if ($file = $this->findFile($class)) {
        includeFile($file);

        return true;
    }
}
```

151.2 RegisterFacades

默认情况下，Laravel 框架包含两个类的自动加载函数，其中一个就是上面所说的 Composer 类自动加载函数，另一个则是在外观注册过程中实现的，也就是在 Illuminate\Foundation\Bootstrap\Regis 类中实现的，其最终结果是把一个类自动加载函数注册到堆栈中，并且在请求到响应的生命周期中进行调用。

RegisterFacades 类的自动加载函数的注册过程也是先实例化 Illuminate\Foundation\Bootstrap\RegisterFacades 类。

```
<?php

namespace Illuminate\Foundation\Bootstrap;

use Illuminate\Support\Facades\Facade;
use Illuminate\Foundation\AliasLoader;
use Illuminate\Contracts\Foundation\Application;

class RegisterFacades
{
    /**
     * Bootstrap the given application.
     *
     * @param \Illuminate\Contracts\Foundation\Application $app
     * @return void
     */
    public function bootstrap(Application $app)
    {
        Facade::clearResolvedInstances();

        Facade::setFacadeApplication($app);

        AliasLoader::getInstance($app->make('config')->get('app.aliases'))->register();
    }
}
```

接下来调用 register() 函数，并在其内部调用 prependToLoaderStack() 方法，最后把 `load($alias)` 函数注册为类的自动加载函数，这个函数的作用主要是将外观别名和外观名（Facades）对应起来，从而实现对应外观类的静态方法调用。

```
<?php
namespace Illuminate\Foundation;

class AliasLoader
{
    protected $aliases;
    protected $registered = false;
    protected static $instance;

    private function __construct($aliases) {
```

```
    $this->aliases = $aliases;
}

public static function getInstance(array $aliases = []) {
    if (is_null(static::$instance)) {
        return static::$instance = new static($aliases);
    }

    $aliases = array_merge(static::$instance->getAliases(), $aliases);
    static::$instance->setAliases($aliases);
    return static::$instance;
}

/*
 * 加载一个类别名(给外观类起一个别名,二者对应一个类)
public function load($alias) {
    if (isset($this->aliases[$alias])) {
        return class_alias($this->aliases[$alias], $alias);
    }
}

/*
* 添加别名到自动加载函数中
*/
public function alias($class, $alias) {
    $this->aliases[$class] = $alias;
}

/*
 * 注册自动加载函数到自动加载堆栈中
 */
public function register() {
    if (! $this->registered) {
        $this->prependToLoaderStack();

        $this->registered = true;
    }
}

/*
 * 把类的自动加载函数添加到自动加载堆栈头部
```

```
     */
    protected function prependToLoaderStack() {
        spl_autoload_register([$this, 'load'], true, true);
    }
}
```

Chapter 152

Anonymous

匿名函数（anonymous 或 closure）是一个没有指定名称的函数，可以用作回调函数的参数。

匿名函数本身是闭包类（Closure）的一个实现，用户可以匿名函数（即闭包对象）创建后对其进行更多的控制，而且匿名函数可以使得框架更加紧凑和灵活。

在匿名函数出现之前，通常调用函数时传入的参数就是数据，只能通过参数对函数的结果进行控制，但是无法控制过程。

匿名函数作为参数传递给函数后，可以作为变量赋值来控制函数的执行过程，因此可以实现更加高效的设计方案。

```
<?php
$array = array(1,2,3,4);
array_walk($array,function($value){echo $value;});
// 输出: 1 2 3 4
```

匿名函数还可以从父作用域中继承变量，即匿名函数在定义时如果需要使用作用域外的变量，可以使用 use 关键字来继承作用域外部的变量。

```
<?php
function getCounter(){
  $i = 0;
  return function() use ($i) {
    echo ++$i;
  }
}
$counter = getCounter();
$counter(); // 1
$counter(); // 1
```

匿名函数在每次执行时都可以访问到上层作用域内部的变量，这些变量在匿名函数被销毁之前始终保存着自己的状态。

上述的示例中，两次函数调用都没有使变量 $i 变量自增，PHP 默认通过复制的方式传入上层变量进入匿名函数，如果需要改变上层变量的值，则需要通过引用的方式传递（即 `use(&$i)`）。

Laravel 框架使用匿名函数完成服务提供者注册等操作。具体来说，Laravel 框架在服务提供者的注册过程中，通过将服务名称和提供服务的匿名函数进行绑定，在使用时可以实现动态服务解析。

这里，可以把所有的服务理解为对一种资源的提供，这个资源可以是一个类的实例、一个路径或一个文件等，因此提供服务就是提供一种资源。

在 Illuminate\Routing\ControllerServiceProvider.php 中，使用 `$this->app->singleton()` 把服务名 illuminate.route.dispatcher 和后面的提供服务的匿名函数进行绑定来提供服务解析，这里的服务就是通过匿名函数实现的。

```
<?php
namespace Illuminate\Routing;
use Illuminate\Support\ServiceProvider;
class ControllerServiceProvider extends ServiceProvider {
   public function register(){
      $this->app->singleton('illuminate.route.dispatcher',function(){
         return new ControllerDispatcher($app['router'],$app);
      });
   }
}
```

Chapter 153

Reflection

PHP 提供了完整的反射 API 来对类、接口、函数、方法和扩展进行反向工程，从而实现动态获取类、对象和方法等语言构件的信息，而且反射 API 还提供方法来取出类、函数和方法中的文档注释。

反射 API 支持对类、对象和方法等语言构件信息的动态获取和动态修改，从而可以使代码更加高效。例如，可以事先不知道需要实例化哪个类，在运行时根据动态信息进行确定，反射机制可以获取实例化类的构造函数信息并完成相应的实例化。

```
<?php
class A {
   public function call() {
      echo 'Hello,world';
   }
}

$ref = new ReflectionClass('A');
$inst = $ref->newInstanceArgs();
$inst->call();
```

如果内置的 PHP 类丢失了反射属性的数据，那么这种情况应该被认为是错误，需要进行修复。

Laravel 框架的服务容器在解析服务的过程中使用反射机制实现依赖注入等操作。例如，Illuminate\Container\Container 类提供的 build() 方法可以根据指定的类来初始化一个具体实例来实现解析服务。

```
public function build($concrete, array $parameters = [])
{
   if ($concrete instanceof Closure) {
      return $concrete($this, $parameters);
```

```
    }
    $reflector = new ReflectionClass($concrete);
    if (! $reflector->isInstantiable()) {
        if (! empty($this->buildStack)) {
            $previous = implode(', ', $this->buildStack);
            $message = "Target [$concrete] is not instantiable while building [$previous].
                ";
        } else {
            $message = "Target [$concrete] is not instantiable.";
        }
        throw new BindingResolutionException($message);
    }
    $this->buildStack[] = $concrete;
    $constructor = $reflector->getConstructor();

    if (is_null($constructor)) {
        array_pop($this->buildStack);
        return new $concrete;
    }
    $dependencies = $constructor->getParameters();
    $parameters = $this->keyParametersByArgument(
        $dependencies, $parameters
    );
    $instances = $this->getDependencies(
        $dependencies, $parameters
    );
    array_pop($this->buildStack);
    return $reflector->newInstanceArgs($instances);
}
```

build() 方法实现解析服务的情况一般分为两种，第一种是查找对应服务是否被服务提供者注册为实例或者提供服务的匿名函数，如果是则直接进行服务解析，第二种是服务名称没有相应的服务绑定，需要通过反射机制来动态创建服务。

反射机制动态创建服务的过程可以分为两个步骤，第一步是通过反射机制获取服务类构造函数的信息，第二步是解决服务类构造函数的依赖问题。

- 通过 `$reflector = new ReflectionClass($concrete);` 来创建一个反射类实例，其中 $concrete 是类的名称。
- 通过 `$reflector->isInstantiable();` 判断这个类是否可以实例化，如果不可以则抛出异常。
- 通过 `$constructor=$reflector->getConstructor();` 来获取类的构造函数，如果

这个类存在构造函数则返回一个 ReflectionMethod 对象（相当于获取构造函数的反射类），如果不存在则返回 NULL。

- 通过 `is_null($constructor)` 判断是否存在构造函数，如果不存在就直接实例化该类，如果存在则通过 `$dependencies = $constructor->getParameters();` 来获取构造函数依赖的输入参数。

Illuminate\Container\Container 的 getDependencies() 方法可以解决构造函数中依赖参数的问题并实现依赖注入，具体来说就是根据反射参数解决所有的参数依赖，同时解决无法获取类名的依赖。

```
//根据反射参数解决所有的参数依赖
protected function getDependencies(array $parameters, array $primitives = [])
{
    $dependencies = [];
    foreach ($parameters as $parameter) {
        $dependency = $parameter->getClass();
        if (array_key_exists($parameter->name, $primitives)) {
            $dependencies[] = $primitives[$parameter->name];
        } elseif (is_null($dependency)) {
            $dependencies[] = $this->resolveNonClass($parameter);
        } else {
            $dependencies[] = $this->resolveClass($parameter);
        }
    }
    return $dependencies;
}
//解决无法获取类名的依赖
protected function resolveNonClass(ReflectionParameter $parameter)
{
    if (! is_null($concrete = $this->getContextualConcrete('$'.$parameter->name))) {
        if ($concrete instanceof Closure) {
            return call_user_func($concrete, $this);
        } else {
            return $concrete;
        }
    }
    if ($parameter->isDefaultValueAvailable()) {
        return $parameter->getDefaultValue();
    }
    $message = "Unresolvable dependency resolving [$parameter] in class {$parameter->
        getDeclaringClass()->getName()}";
    throw new BindingResolutionException($message);
```

```
}
//通过服务容器解决一个具有类名的依赖
protected function resolveClass(ReflectionParameter $parameter)
{
    try {
        return $this->make($parameter->getClass()->name);
    } catch (BindingResolutionException $e) {
        if ($parameter->isOptional()) {
            return $parameter->getDefaultValue();
        }
        throw $e;
    }
}
```

依赖和依赖注入可以理解为获取类构造函数中的参数并完成类的实例化的过程。

- 通过 `$parameters=$this->keyParametersByArgument($dependencies,$parameters);` 获取直接提供的实参,未直接提供的通过 `$instances = $this->getDependencies($dependencies` 根据形参的类型获取实参。
- 在 getDependencies() 函数中需要调用 resolveNonClass() 函数或 resolveClass() 函数解决参数依赖问题。
  对于构造函数的参数，如果无法获取该参数的类型，则通过 resolveNonClass(0 函数获取默认的参数值，如果可以获取类的名称，则通过 resolveClass(0 函数进行实例化。
- 实例化过程是通过服务容器进行解析的，即通过 `$parameter->getClass()->name` 获取参数的类名，然后通过 `$this->make($parameter->getClass()->name;` 来解析服务。
- make() 函数接下来还会调用 build() 函数完成类的实例化过程，其实质相当于一个递归调用的过程，并最终由 `$reflector->newInstanceArgs($instance);` 实例化服务类，进而完成服务的解析。

Chapter 154

Trait

PHP 和 Ruby 只支持单一继承，因此相对于夺继承语言（例如 C++）来说需要通过其他办法来解决代码复用问题。

- PHP 的 Trait 以水平特性的组合来实现代码复用
- Ruby 的混入类（Mixin）可以实现代码复用。

trait 和类相似，只是 trait 无法被实例化，需要通过关键词 use 添加到其他类的内部来发挥作用，因此 trait 使用的是不同于传统继承方法的水平特性组合。

```
<?php
class Base {
   public function hello() {
      echo 'method hello from base class';
   }
}
trait Hello {
   public function hello() {
       echo 'method hello from trait Hello.';
   }
   public function hi() {
      echo 'method hi from trait Hello';
   }
   abstract public function getValue();
   static public function staticMethod() {
      echo 'static method staticMethod from trait Hello';
   }
   public function staticValue() {
      static $value;
      $value++;
```

```
      echo $value;
   }
}
trait Hi {
   public function hello() {
       parent::hello();
       echo 'method hello from trait Hi';
   }
   public function hi() {
      echo 'method hi from trait Hi';
   }
}
trait HelloHi {
   use Hello,Hi {
      Hello::hello insteadof Hi;
      Hi::hi insteadof Hello;
   }
}
class MyNew extends Base {
   use HelloHi;
   private $value = 'class MyNew';
   public function hi() {
       echo 'method hi from class MyNew';
   }
   public function getValue() {
       return $this->value;
   }
}
$obj = new MyNew();
// trait的方法可以覆盖基类的方法
$obj->hello();// method hello from trait Hello

// 当前类中的方法可以覆盖trait的方法
$obj->hi();// method hi from class MyNew

// trait可以定义静态方法
MyNew::staticMethod();//static method staticMethod from trait Hello

//trait可以使用抽象方法
echo $obj->getValue();// class MyNew
```

```
//trait可以使用静态成员
$objOther = new MyNew();
$obj->staticValue();// 1
$objOther->staticValue();// 2
```

- 当前类的方法会覆盖 trait 中的方法，trait 中的方法会覆盖基类的方法。
- 多个 trait 使用逗号分隔并通过 use 关键字列出多个 trait。
- 如果两个 trait 都插入了一个同名的方法，如果没有明确解决冲突就会产生一个致命错误。
- 使用 as 语法调整方法的访问控制。
- 在 trait 中可以使用抽象成员来强制类中必须实现这个抽象方法。
- 在 trait 中可以使用静态方法和静态变量。
- 在 trait 中可以定义属性。

为了解决多个 trait 在同一个类中的命名冲突，需要使用 insteadof 操作符来明确指定使用冲突方法中的哪一个。或者，可以使用 as 操作符将其中一个冲突的方法以另一个名字来引入。

Laravel 使用 trait 来实现代码复用（例如身份认证）。

Part XXXII

Deployment

Chapter 155

Overview

155.1 Platform

部署 PHP 应用程序到生产环境中有多种方式。例如，PaaS（例如 Heroku、Openshift、Beanstalk、Azure 和 Google App Engine 等）提供了运行 PHP 应用程序所必须的系统环境和网络架构，只需做少量配置就可以运行 PHP 应用程序或者 PHP 框架。

155.2 FastCGI

PHP 通过内置的 FastCGI 进程管理器（FPM），可以很好的与轻量级的高性能 web 服务器 nginx 协作使用。nginx 比 Apache 占用更少内存而且可以更好的处理并发请求，这对于并没有太多内存的虚拟服务器尤其重要。

PHP 和 Apache 有很长的合作历史。Apache 有很强的可配置性和大量的扩展模块，完美支持各种 PHP 框架和开源应用 (如 WordPress)。

默认情况下，mod_php5 必须使用 prefork MPM，因此 Apache 会比 nginx 消耗更多的资源，而且并发处理能力不强。

Apache 有多种方式运行 PHP，最常见的方式就是使用 mod_php5 的 prefork MPM 方式，但是它对内存的利用效率并不高，如果并不想深入服务器管理方面，那么可以部署为 prefork MPM 方式。

如果追求高性能和高稳定性，可以为 Apache 选择与 nginx 类似的的 FPM 系统 worker MPM 或者 event MPM，它们分别使用 mod_fastcgi 和 mod_fcgid，可以更高效的利用内存，运行速度也更快，同时配置也相对复杂一些。

在共享主机上部署 PHP 应用程序时，需要注意服务器上的 PHP 是否是最新稳定版本。共享主机允许多个开发者把自己的网站部署在上面，这样的好处是费用非常便宜，坏处是无

法知道将和哪些网站共享主机，因此需要仔细考虑机器负载和安全问题。如果项目预算允许的话，避免使用共享主机是上策。

实际上，只有虚拟服务器或者专用服务器才允许开发者完全控制自己的生产环境。如果在手动进行数据库结构的修改或者在更新文件前手动运行测试，每一个额外的手动任务的添加都需要去部署一个新的版本到应用程序，这些更改会增加程序潜在的致命错误。即便只是处理一个简单的更新，全面的构建处理或者持续集成策略来实现构建自动化都会带来好处。

155.3 Cache

PHP 本身来说是非常快的，但是在发起远程连接、加载文件等操作时也会遇到瓶颈，不过已经有各种各样的工具可以用来加速 PHP 应用程序的某些耗时的部分，或者说减少某些耗时任务所需要运行的次数。

155.3.1 Opcode

当一个 PHP 文件被解释执行的时候，首先是被编译成名为 opcode 的中间代码，然后才被底层的虚拟机执行。如果 PHP 文件没有被修改过，opcode 始终是一样的，因此每次从头编译 PHP 到 opcode 就意味着编译步骤白白浪费了 CPU 的资源。

PHP 的 opcode 缓存工具可以把 opcode 缓存在内存中，它能防止冗余的编译步骤，并且在下次调用执行时得到重用，结果就是让 PHP 应用程序大大加速。

155.3.2 Object

有时缓存代码中的单个对象会很有用，比如有些需要很大开销获取的数据或者一些结果集不怎么变化的数据库查询，可以使用一些缓存软件将这些数据存放在内存中以便下次高速获取。如果获得数据后把它们储存到对象缓存服务器中，下次请求直接从缓存里面获取数据，在减少数据库负载的同时能极大提高性能。

另外，许多流行的字节码缓存方案也能缓存定制化的数据，例如 APCu、XCache 以及 WinCache 都提供了 API 来将数据缓存到内存中。

在实际应用环境中，最常用的内存对象缓存系统是 APCu 和 Memcached 。APCu 对于对象缓存来说是个很好的选择，它提供了简单的 API 来将数据缓存到内存，并且很容易设置和使用。APCu 的局限性表现在它依赖于所在的服务器。另一方面，Memcached 和 Redis 等可以以独立的服务的形式安装，可以通过网络交互，这意味着可以将数据集中存在一个高速存取的地方，而且许多不同的系统能从中获取数据。

值得注意的是，当用户以 CGI（FastCGI）的形式使用 PHP 时，每个进程将会有各自的缓存，比如说 APCu 缓存数据无法在多个工作进程中共享。在这种情况下，可能就必须考虑

Memcached 或 Redis，因为它们可以独立于 PHP 进程。

通常情况下，APCu 在存取速度上比 Memcached 更快，但是 Memcached 在扩展上更有优势。如果不希望应用程序涉及多个服务器，或者不需要 Memcached 提供的其他特性，那么 APCu 可能是最好的选择。

```
<?php
// check if there is data saved as 'expensive_data' in cache
$data = apc_fetch('expensive_data');
if($data === false) {
  // data is not in cache
  // save result of expensive call for later use
  apc_add('expensive_data',$data = get_expensive_data());
}
print_r($data);
```

APC 同时提供了对象缓存与字节码缓存，APCu 是为了将 APC 的对象缓存移植到 PHP 5.5+ 的一个项目，因为现在 PHP 有了内建的字节码缓存方案 (OPcache)。

Chapter 156

Deployment

在部署和维护 PHP 应用程序时，可能需要实现自动化的任务包括依赖管理（Composer）、静态资源编译和压缩、执行测试、文档生成（PHPDoc）、打包（Phar）和部署等。

构建工具可以认为是一系列的脚本来完成应用部署的通用任务。构建工具并不属于应用的一部分，它独立于应用层‘之外’。

现在已有很多开源的工具来帮助你完成构建自动化，一些是用 PHP 编写，有一些不是。应该根据实际项目来选择最适合的工具，不要让语言阻碍了这些工具的使用。

- Phing
- Capistrano
- Chef
- Deployer

156.1 Chef

Chef 不仅仅只是一个部署框架，它是一个基于 Ruby 的强大的系统集成框架，除了部署应用之外，还可以构建整个服务环境或者虚拟机。

156.2 PHPCI

156.3 Phing

Phing 是一种在 PHP 领域中最简单的开始自动化部署的方式。通过 Phing 可以控制打包、部署或者测试，只需要一个简单的 XML 构建文件。

Phing (基于 Apache Ant) 提供了在安装或者升级 web 应用时的一套丰富的任务脚本，并且可以通过 PHP 编写额外的任务脚本来扩展。

156.4 Travis

持续集成是一种软件开发实践，团队的成员经常用来集成他们的工作，通常每一个成员至少每天都会进行集成—因此每天都会有许多的集成。许多团队发现这种方式会显著地降低集成问题，并允许一个团队更快的开发软件。

PHP 有许多方式可以来实现持续集成。例如，近来 Travis CI 在持续集成上做的很棒，对于小项目来说也可以很好的使用。

Travis CI 是一个托管的持续集成服务用于开源社区，而且 Travis CI 可以和 Github 很好的集成，并且提供了很多语言的支持（包括 PHP）。

156.5 Jenkins

156.6 Deployer

Deployer 是一个用 PHP 编写的部署工具，它很简单且实用，可以并行执行任务，原子化部署，以及在多台服务器之间保持一致性。

Deployer 为 Symfony、Laravel、Zend Framework 和 Yii 提供了通用的任务脚本。

156.7 Capistrano

Capistrano 以一种结构化、可复用的方式在一台或多台远程机器上执行命令。对于部署 Ruby on Rails 的应用，它提供了预定义的配置，不过也可以用它来部署 PHP 应用。如果要成功的使用 Capistrano ，需要一定的 Ruby 和 Rake 的知识。

Chapter 157

Environment

在开发和线上阶段使用不同的系统运行环境可能会遇到各种各样的 BUG，并且在团队开发时让所有成员都保持使用最新版本的软件和类库也是一件很让人头痛的事情.

如果在 Windows 下开发并部署到 Linux 环境中，或者团队协同开发时，建议使用虚拟机。

除了 VMware 和 VirtualBox 外, 还可以使用 Vagrant 和 Docker 等实现快速切换到虚拟环境。

157.1 Vagrant

Vagrant 可以使用单一的配置信息来部署一套虚拟环境，最后打包为一个所谓的 box (就是已经部署好环境的虚拟机器)，而且可以手动安装和配置 box 或者使用自动部署工具（例如 Puppet 或者 Chef)。

自动部署工具的好处就是可以快速部署一套一模一样的环境，避免了一大堆的手动的命令输入，并且允许用户随时删除和重建一个全新的 box，从而简化虚拟机的管理。

Vagrant 还可以在虚拟机和主机上分享文件夹，这样就可以在主机里面编辑代码，然后在虚拟机里面运行。

下面的一些工具可以用来更好地使用 Vagrant:

- Rove 使用 Chef 自动化安装一些常用的软件（包括 PHP)。
- Puphpet 使用 Web 图形界面来生成部署 PHP 环境的 Puppet 脚本。
- Protobox 是一个基于 vagrant 的一个层，同时还有 Web 图形界面，而且允许使用 YAML 文件来安装和配置虚拟机里面的软件。
- Phansible 提供了一个简单的 Web 图形界面来创建 Ansible 自动化部署脚本，专门为 PHP 项目定制。

157.2 Container

Docker 为各种应用程序提供了 Linux 容器，因此 Docker 是除了 Vagrant 之外另一个实现生产和开发环境统一的虚拟化方案。

用户可以安装 Docker 镜像（例如 MySQL 和 PostgreSQL 等），并且不会污染到本地机器。例如，下面的命令会下载一个功能齐全的 Apache 和最新版本的 PHP，并且设置 WEB 目录 /path/to/your/php/files 运行在 http://localhost:8080。

```
$ docker run -d --name my-php-webserver -p 8080:80 -v /path/to/your/php/files:/var/www/
    html/ php:apache
```

在使用 docker run 命令以后停止或者再次开启容器，只需要执行以下命令：

```
$ docker stop my-php-server
$ docker start my-php-server
```

Chapter 158

Documentation

PHPDoc 是注释 PHP 代码的非正式标准，PHPDoc 提供了许多不同的标记可以使用。

```
<?php
/**
 * @author A Name <a.name@example.com>
 * @link http://www.phpdoc.org/docs/latest/index.html
 */
class DateTimeHelper
{
    /**
     * @param mixed $anything Anything that we can convert to a \DateTime object
     *
     * @throws \InvalidArgumentException
     *
     * @return \DateTime
     */
    public function dateTimeFromAnything($anything)
    {
        $type = gettype($anything);

        switch ($type) {
            // Some code that tries to return a \DateTime object
        }

        throw new \InvalidArgumentException(
            "Failed Converting param of type '{$type}' to DateTime object"
        );
    }
```

```
    /**
     * @param mixed $date Anything that we can convert to a \DateTime object
     *
     * @return void
     */
    public function printISO8601Date($date)
    {
        echo $this->dateTimeFromAnything($date)->format('c');
    }

    /**
     * @param mixed $date Anything that we can convert to a \DateTime object
     */
    public function printRFC2822Date($date)
    {
        echo $this->dateTimeFromAnything($date)->format('r');
    }
}
```

上述这个类的说明中使用了 @author 和 @link 标记，其中 @author 标记是用來说明代码的作者，在多位开发者的情况下可以同时列出好几位，其次 @link 标记用来提供网站链接来进一步说明代码和网站之间的关系。

在这个类中，第一个方法的 @param 标记，说明类型、名字和传入方法的参数。

此外，@return 和 @throws 标记说明返回类型以及可能抛出的异常。

第二、第三个方法非常类似，和第一个方法一样使用一个 @param 标记。第二、和第三个方法之间关键差别在注释区块使用/排除 @return 标记。@return void 标记明确告诉我们没有返回值，而过去省略 @return void 声明也具有相同效果（没有返回任何值）。

Part XXXIII

Security

Chapter 159

Overview

绝对安全的系统是不存在的，因此安全业界常用的方法有助于平衡可用性和风险。例如，对用户提交的每一个变量都进行双重验证可能是一个很负责任的行为，不过同时也会导致用户必须花很多时间去填写一张复杂无比的表格，这样就会让某些用户尝试去绕过安全机制。

最好的安全机制应该能在不防碍用户，并且不过多地增加开发难度的情况下做到能满足需求，而且过度强化安全机制的系统上也可能出现某些问题。

一个系统的的强度是由它最薄弱的环节决定的。例如，如果所有的事务都基于时间、地点、事务种类进行详细的记录，但是用户验证却只依靠一个 cookie，那么用户所对应的事务记录的可信度就被大大剥弱了。

调试代码时也要事先明白就算是一个简单的页面也很难对所有可能发生的情况进行检测，黑客有足够的时间研究系统，因此必须检查所有的代码去发现哪里可以引入不正当的数据，然后对代码改进、简化或者增强。

PHP 解释器放在 Web 目录外的地方无疑是一个非常安全的做法，这样做唯一不便的地方就是必须在每一个包含 PHP 代码的文件的第一行加入如下语句，还要将这些文件的属性改成可执行。

```
#!/usr/local/bin/php
<?php
```

PHP 解释器移出 Web 目录后，在编译 PHP 解释器时必须加入`--enable-discard-path`参数才能正确处理 PATH_INFO 和 PATH_TRANSLATED 等变量。

除了把 PHP 以模块的形式嵌入到 Apache 等服务器内部之外，还可以以 CGI 模式安装 PHP，那么应该做到的安全措施包括如下：

- 只运行公开的文件
- 使用`--enable-force-cgi-redirect`选项
- 设置 doc_root 或 user_dir

- PHP 解释器放在 web 目录以外

实际上，还可以把 PHP 用于不同的 CGI 封装来为代码创建安全的 chroot 和 setuid 环境。

通常情况下，用户可能会把 PHP 的可执行文件安装到 Web 服务器的 cgi-bin 目录，现在建议不要把任何的解释器放到 cgi-bin 目录。

如果以类似 http://my.host/cgi-bin/php?/etc/passwd 的方式来访问系统文件，那么在 URL 请求的问号（?）后面的信息会传给 CGI 接口作为命名行的参数。

其它的解释器可能会在命令行中打开并执行第一个参数所指定的文件，不过以 CGI 模式安装的 PHP 解释器被调用时会拒绝解释这些参数。

如果以类似 http://my.host/cgi-bin/php/secret/doc.html 的方式来访问服务器上的任意目录，PHP 解释器所在目录后面的 URL 信息 /secret/doc.html 将会例行地传给 CGI 程序并进行解释。

一些 Web 服务器会重定向到页面 (例如如 http://my.host/secret/script.php), 那么这些服务器就会先检查用户是否有访问 /secret 目录的权限, 然后创建 http://my.host/cgi-bin/php/secret/script.php 上的页面重定向。

实际情况是很多服务器并不会检查用户访问 /secret/script.php 的权限，仅仅只检查了 /cgi-bin/php 的权限，这样任何能访问 /cgi-bin/php 的用户就可以访问 web 目录下的任意文件了。

用户可以编译 PHP 时配置选项`--enable-force-cgi-redirect` 以及运行时配置指令 doc_root 和 user_dir 来对服务器上的文件和目录添加限制以防止这类攻击。

`--enable-force-cgi-redirect` 编译选项可以防止任何人通过请求 http://my.host/cgi-bin/php/secretdir/script.php 这样的 URL 来直接调用 PHP，PHP 在此模式下只会解析已经通过了 Web 服务器的重定向规则的 URL。

如果 Web 服务器中所有内容都受到密码或 IP 地址的访问限制，就不需要设置上述这些选项。如果 Web 服务器不支持重定向，或者 Web 服务器不能和 PHP 通信都可以使访问请求变得更为安全，可以在 configure 脚本中指定`--enable-force-cgi-redirect` 选项。

除此之外, 用户还要确认 PHP 程序不依赖其它方式调用, 例如通过直接访问 http://my.host/cgi-bin/php/dir/script.php 访问或通过重定向访问 http://my.host/dir/script.php。

在 Apache 中，重定向可以使用 AddHandler 和 Action 语句来设置。

```
Action php-script /cgi-bin/php
AddHandler php-script .php
```

Apache 重定向依赖于非标准 CGI 环境变量 REDIRECT_STATUS，因此如果 Web 服务器不支持任何方式能够判断请求是直接的还是重定向的，就不能使用这个选项，而是应该考虑用其它方法。

在 Web 服务器的主文档目录中包含动态内容 (例如脚本和可执行程序) 有时被认为是一种不安全的实践。如果因为配置上的错误而未能执行脚本而作为普通 HTML 文档显示，很可能导致敏感信息泄露。

为了安全性考虑，应该专门设置一个只能通过 PHP CGI 来访问的目录，这样该目录中的内容只会被解析而不会原样显示出来。

对于无法判断是否重定向的情况，很有必要在主文档目录之外建立一个专用于脚本的 doc_root 目录。例如，PHP 的配置文件中提供了 doc_root 或设置环境变量 PHP_DOCUMENT_ROOT 来自定义 PHP 脚本主目录，这样 PHP 就只解释 doc_root 目录下的文件，从而确保目录外的脚本不会被 PHP 解释器执行（user_dir 除外）。

如果未设置 user_dir，doc_root 就是唯一能控制在哪里打开文件的选项。例如，访问 `http://my.host/~user/doc.php` 并不会打开用户主目录下文件，而只会执行 doc_root 目录下的 `~user/doc.php`（这个子目录以 [~] 开头）。

如果设置了 user_dir（例如 public_php），那么类似 `http://my.host/~user/doc.php` 的请求就会执行用户主目录下的 public_php 子目录下的 doc.php 文件。如果用户主目录的绝对路径是 /home/user，那么被执行文件将会是 /home/user/public_php/doc.php。

user_dir 的设置与 doc_root 无关，可以用来分别控制 PHP 脚本的主目录和用户目录。

某些情况下，尽可能的多增加一份安全性都是值得的。例如，使用隐藏 PHP 的手段提高安全性被认为是作用不大但是也值得的做法，或者在 php.ini 文件里设置 `expose_php = off` 也可以减少攻击者能获得的有用信息。

Web 服务器可以配置成使用 PHP 解析不同的扩展名，通过.htaccess 文件和 Apache 的配置文件都可以设置误导攻击者的文件扩展名：

```
AddType application/x-httpd-php .asp .py .pl
```

Example 153　使用未知的扩展名作为 PHP 的扩展名

```
AddType application/x-httpd-php .bop .foo .133t
```

或者，可以把 PHP 文件隐藏为 HTML 页面，这样所有的 HTML 文件都会通过 PHP 引擎，会为服务器增加一些负担。

```
AddType application/x-httpd-php .htm .html
```

虽然把 PHP 文件的扩展名改为以上的扩展名可以通过隐藏来提高安全性，但是防御能力仍然很低而且有缺点。

159.1 Apache Permission

如果以 Apache 模块的方式安装 PHP，那么 PHP 将继承 Apache 用户（通常为“nobody”）的权限，进而对安全和认证有一些影响。比如，如果用 PHP 来访问数据库，除非数据库有自己的访问控制，否则就要使“nobody”用户可以访问数据库，这样就意味着恶意的脚本在

不用提供用户名和密码时就可能访问和修改数据库，而且 Web Spider 也完全有可能偶然发现数据库的管理页面，并且删除所有的数据库。

用户可以通过 Apache 认证来避免权限问题，或者用 LDAP、.htaccess 等技术来设计自己的访问模型，并把这些代码作为 PHP 脚本的一部份。

通常情况下，在安全性达到可以使 PHP 用户（这里也就是 Apache 用户）承担的风险极小的程度时候，可能 PHP 已经到了阻止向用户目录写入任何文件或禁止访问和修改数据库的程度。也就是说，无论是正常的文件还是非正常的文件，无论是正常的数据库事务来是恶意的请求都会被拒之门外。

另外一个常犯的对安全性不利的错误就是让 Apache 拥有 root 权限，或者通过其它途径赋予 Apache 更强大的功能。

必须注意的是，把 Apache 用户的权限提升为 root 是极度危险的做法，而且可能会危及到整个系统的安全，所以不要考虑使用 su，chroot 或者以 root 权限运行 Apache。

除此之外，还可以使用 open_basedir 来限制哪些目录可以被 PHP 使用，也可以设置 Apache 的专属区域来把所有的 Web 活动都限制到非用户和非系统文件中。

159.2 Session Management

保持 HTTP 会话管理的安全是重要的。

159.3 Filesystem Permission

PHP 遵从大多数服务器系统中关于文件和目录权限的安全机制，因此可以控制哪些文件在文件系统内是可读的。必须特别注意的是全局的可读文件，并确保每一个有权限的用户对这些文件的读取动作都是安全的。

PHP 被设计为以用户级别来访问文件系统，所以完全有可能通过编写 PHP 代码来读取系统文件（例如/etc/passwd），更改网络连接以及发送海量打印任务等等，因此必须确保 PHP 代码读取和写入的是合适的文件。

例如，用户需要删除自己主目录中的一个文件时，如果通过 Web 界面来管理文件系统，那么 Apache 用户就有权删除用户目录下的文件。

Example 154 应该对变量进行安全检查

```
<?php
// 从用户目录中删除指定的文件
$username = $_POST['user_submitted_name'];
$userfile = $_POST['user_submitted_filename'];
$homedir = "/home/$username";
unlink ("$homedir/$userfile");
```

```
echo "The file has been deleted!";
```

如果 username 和 filename 变量可以通过用户表单来提交，那么就可以提交别人的用户名和文件名来删掉他人的文件。这里，如果提交的变量是“../etc/”和“passwd”就会导致无法登录系统，因此需要考虑其它方式的认证。

- 仅赋予 PHP 的 Web 用户有限的权限。
- 必须检查用户提交的所有变量。

Example 155　构造文件系统攻击

```
<?php
// 删除硬盘中任何 PHP 有访问权限的文件。如果 PHP 有 root 权限:
$username = $_POST['user_submitted_name']; // "../etc"
$userfile = $_POST['user_submitted_filename']; // "passwd"
$homedir = "/home/$username"; // "/home/../etc"

unlink("$homedir/$userfile"); // "/home/../etc/passwd"

echo "The file has been deleted!";
```

Example 156　更安全的文件名检查

```
<?php
// 删除硬盘中 PHP 有权访问的文件
$username = $_SERVER['REMOTE_USER']; // 使用认证机制
$userfile = basename($_POST['user_submitted_filename']);
$homedir = "/home/$username";

$filepath = "$homedir/$userfile";


if (file_exists($filepath) && unlink($filepath)) {
    $logstring = "Deleted $filepath\n";
} else {
    $logstring = "Failed to delete $filepath\n";
}
$fp = fopen("/home/logging/filedelete.log", "a");
fwrite ($fp, $logstring);
fclose($fp);

echo htmlentities($logstring, ENT_QUOTES);
```

即使进行变量过滤和权限检查，仍然还是有缺陷。例如，如果认证系统允许用户建立自己的登录用户名，而用户选择用“../etc/”作为用户名，那么攻击还是可以实现的。

Example 157 改进的更安全的文件名检查

```
<?php
$username = $_SERVER['REMOTE_USER']; // 使用认证机制
$userfile   = $_POST['user_submitted_filename'];
$homedir = "/home/$username";

$filepath   = "$homedir/$userfile";
if (!ctype_alnum($username) || !preg_match('/^(?:[a-z0-9_-]|\.(?!\.))+$/iD',
     $userfile)) {
   die("Bad username/filename");
}
```

不同的操作系统存在着各种不同的需要注意的文件，包括系统设备（/dev/ 或者 COM1）、配置文件（/etc/和.ini 文件）、常用的存储区域（/home/ 或者 My Documents）等，因此更应该建立一个禁止所有权限而只开放明确允许的资源的策略。

159.4 Null bytes issue

PHP 的文件系统操作是基于 C 语言函数，所以 PHP 可能会以用户意想不到的方式处理 Null 字符。例如，Null 字符在 C 语言中用于标识字符串结束，一个完整的字符串是从其开头到遇见 Null 字符为止。

Example 158 Null 字符攻击

```
<?php
$file = $_GET['file']; // "../../etc/passwd\0"
if (file_exists('/home/wwwrun/'.$file.'.php')) {
   // file_exists will return true as the file /home/wwwrun/../../etc/passwd
        exists
   include '/home/wwwrun/'.$file.'.php';
   // the file /etc/passwd will be included
}
```

任何用于操作文件系统的字符串（特别是程序外部输入的字符串）都必须经过适当的检查。

Example 159 正确验证用户输入

```php
<?php
$file = $_GET['file'];

// 对字符串进行白名单检查
switch ($file) {
    case 'main':
    case 'foo':
    case 'bar':
        include '/home/wwwrun/include/'.$file.'.php';
        break;
    default:
        include '/home/wwwrun/include/main.php';
}
```

159.5 Database Attack

从数据库中提取或者存入数据必须经过连接数据库、发送一条合法查询、获取结果、关闭连接等步骤，攻击者可以通过篡改 SQL 查询语句实现 SQL 注入攻击。

数据库安全的原则就是深入防御，虽然 PHP 本身并不能保护数据库的安全，不过可以利用 PHP 脚本对数据库进行基本的访问和操作，而且保护数据库的措施越多就越难让攻击者获得和使用数据库内的信息。

正确地设计和应用数据库同样也可以减少被攻击的担忧。

159.5.1 Design Database

在设计数据库时，除非是使用第三方的数据库服务，否则应用程序永远不要使用数据库所有者或超级用户帐号来连接数据库，因为这些帐号可以执行任意的操作，比如说修改数据库结构（例如删除一个表）或者清空整个数据库的内容。

用户应该为程序的每个方面创建不同的数据库帐号，并赋予对数据库对象的极有限的权限。例如，仅分配给能完成其功能所需的权限，避免同一个用户可以完成另一个用户的事情，这样即使攻击者利用程序漏洞取得了数据库的访问权限，也最多只能做到和该程序一样的影响范围。

鼓励用户不要把所有的事务逻辑都通过 Web 应用程序（即用户的脚本）来实现，最好用视图（view）、触发器（trigger）或者规则（rule）在数据库层面完成，否则当系统升级的时候，就需要为数据库开辟新的接口，同时必须重做所有的数据库客户端。

除此之外，触发器还可以透明和自动地处理字段，而且可以在调试程序和跟踪事实时提供有用的信息。

159.5.2 Connect Database

把连接建立在 SSL 加密技术上可以增加客户端和服务器端通信的安全性，或者 SSH 也可以用于加密客户端和数据库之间的连接。如果使用了这些技术的话，攻击者要监视服务器的通信或者得到数据库的信息是很困难的。

159.5.3 Encryption/Decryption

虽然 SSL/SSH 可以保护客户端和服务器端交换的数据，但是 SSL/SSH 并不能保护数据库中已有的数据，毕竟 SSL 只是一个加密网络数据流的协议。

如果攻击者取得了直接访问数据库的许可（绕过 Web 服务器），敏感数据就可能暴露或者被滥用（除非数据库自己保护了这些信息）。对数据库内的数据加密是减少这类风险的有效途径，但是只有很少的数据库提供这些加密功能。

对于存储模型的加密问题，简单的解决办法就是创建自己的加密机制，然后将其应用到 PHP 程序中。例如，PHP 提供了相关的扩展库（例如 Mcrypt 和 Mhash 等）来实现加密。

Mcrypt 和 Mhash 包含多种加密运算法则，这样 PHP 脚本可以在插入数据库之前先把数据加密，以后提取出来时再解密。

对某些真正隐蔽的数据，如果不需要以明文的形式存在（即不用显示），可以考虑用散列算法。使用散列算法最常见的例子就是把密码经过 MD5 加密后的散列存进数据库来代替原来的明文密码。

Example 160 对密码字段进行散列加密

```
<?php
// 存储密码散列
$query = sprintf("INSERT INTO users(name,pwd) VALUES('%s','%s');",
          pg_escape_string($username), md5($password));
$result = pg_query($connection, $query);

// 发送请求来验证用户密码
$query = sprintf("SELECT 1 FROM users WHERE name='%s' AND pwd='%s';",
          pg_escape_string($username), md5($password));
$result = pg_query($connection, $query);

if (pg_num_rows($result) > 0) {
   echo 'Welcome, $username!';
} else {
   echo 'Authentication failed for $username.';
}
```

159.5.4 SQL Injection

篡改 SQL 查询可以绕开访问控制、身份验证和权限检查，并且可以通过 SQL 查询去运行主机操作系统级的命令。

创建或修改已有 SQL 语句可以实现直接 SQL 注入，从而可以获取到敏感数据或者覆盖关键的数据，以及执行数据库主机操作系统命令。

攻击者可以通过应用程序取得用户输入并与静态参数组合成 SQL 查询来实现 SQL 注入。例如，在缺乏对输入的数据进行验证，并且使用了超级用户或其它有权创建新用户的数据库帐号来连接数据库时，攻击者可以在数据库中新建一个超级用户。

Example 161 在数据分页显示的代码中构造 SQL 注入创建超级用户

```
<?php
$offset = $argv[0]; // 注意，没有输入验证！
$query = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET
    $offset;";
$result = pg_query($conn, $query);
```

原本代码默认只会认为 $offset 是一个数值，但是攻击者可以通过把以下语句经过 urlencode() 处理后加入 URL 来实现创建超级用户的操作：

```
0;
insert into pg_shadow(usename,usesysid,usesuper,usecatupd,passwd)
   select 'crack', usesysid, 't','t','crack'
   from pg_shadow where usename='postgres';
--
```

注意，上述攻击示例代码中的 `0;` 只不过是为了提供一个正确的偏移量以便补充完整原来的查询来确保 SQL 不要出错，SQL 的注释标记--可以使用来指示 SQL 解释器忽略后面的语句。

从显示搜索结果的页面入手是一个能得到密码的可行办法，攻击者所要做的只不过是找出哪些提交上去的变量是用于 SQL 语句并且处理不当的，而且通常这类的变量都被用于 SELECT 查询中的条件语句（例如 WHERE, ORDER BY, LIMIT 和 OFFSET）。

如果数据库支持 UNION 构造，攻击者还可能会把一个完整的 SQL 查询附加到原来的语句上以便从任意数据表中得到密码，因此对密码字段加密是很重要的。

Example 162 获取用户密码

```
<?php
$query = "SELECT id, name, inserted, size FROM products
                WHERE size = '$size'
                ORDER BY $order LIMIT $limit, $offset;";
$result = odbc_exec($conn, $query);
```

例如，可以在原来的查询的基础上添加另一个 SELECT 查询来获得密码：

```
'
union select '1', concat(uname||'-'||passwd) as name, '1971-01-01', '0' from usertable;
--
```

如果上述语句（使用' 和 --）被加入到 $query 中的任意一个变量中，就可能会成功取得敏感数据。

同样地，SQL 中的 UPDATE 也会受到攻击，虽然 UPDATE 查询也可能被插入或附加上另一个完整的请求，不过攻击者往往更愿意对 SET 子句入手，这样就可以更改数据表中的一些数据。

在使用 SET 子句进行攻击时，攻击者必须事先知道数据库的结构才能修改查询成功进行，但是这些都可以通过表单上的变量名对字段进行猜测，或者进行暴力破解，而且实际上对于存放用户名和密码的字段，命名的方法并不多。

```
<?php
$query = "UPDATE usertable SET pwd='$pwd' WHERE uid='$uid';";
```

恶意的用户会把' or uid like'%admin%'; --作为变量的值提交给 $uid 来改变 admin 的密码，或者把 $pwd 的值提交为 "password', admin='yes', trusted=100 "（后面有个空格）来获得更多的权限。

Example 163 通过重置密码实现攻击

```
<?php
// $uid == ' or uid like'%admin%'; --
$query = "UPDATE usertable SET pwd='...' WHERE uid='' or uid like '%admin%';
     --";

// $pwd == "hehehe', admin='yes', trusted=100 "
$query = "UPDATE usertable SET pwd='hehehe', admin='yes', trusted=100 WHERE
    ...;";
```

```
<?php
$query = "SELECT * FROM products WHERE id LIKE '%$prod%'";
$result = mssql_query($query);
```

如果攻击提交 a%' exec master..xp_cmdshell 'net user test testpass /ADD' --作为变量 $prod 的值，那么 $query 将可以用来攻击数据库主机的操作系统。

Example 164 攻击数据库主机的操作系统

```
<?php
$query = "SELECT * FROM products
                WHERE id LIKE '%a%'
                exec master..xp_cmdshell 'net user test testpass /ADD'--";
$result = mssql_query($query);
```

MSSQL 服务器会执行上述这条 SQL 语句，包括它后面那个用于向系统添加用户的命令。如果这个程序是以 sa 运行而 MSSQLSERVER 服务又有足够的权限的话，攻击者就可以获得一个系统帐号来访问主机。

针对某一特定的数据库系统的攻击可以遵循类似的原理来构造对其它数据库系统实施类似的攻击。

永远不要信任外界输入的数据，特别是来自于客户端的，包括选择框、表单隐藏域和 cookie，攻击者使用正常的查询也有可能给服务器造成灾难。

- 永远不要使用超级用户或所有者帐号去连接数据库，必须使用用权限被严格限制的帐号。
- 检查输入的数据是否具有所期望的数据格式，例如使用变量函数和字符类型函数（比如 is_numeric()，ctype_digit()）到复杂的 Perl 兼容正则表达式函数来进行格式校验。
- 如果程序等待输入一个数字，可以考虑使用 is_numeric() 来检查，或者直接使用 settype() 来转换类型，也可以用 sprintf() 格式化为数字。
- 使用数据库特定的敏感字符转义函数(比如 mysql_escape_string() 和 sql_escape_string())把用户提交上来的非数字数据进行转义。如果数据库没有专门的敏感字符转义功能的话 addslashes() 和 str_replace() 可以代替完成这个工作。

仅在查询的静态部分加上引号是不够的，查询很容易被攻破。

Example 165　安全地实现分页

```
<?php
settype($offset, 'integer');
$query = "SELECT id, name FROM products ORDER BY name LIMIT 20 OFFSET
    $offset;";

// 注意格式字符串中的 %d, 如果用 %s 就毫无意义了
$query = sprintf("SELECT id, name FROM products ORDER BY name LIMIT 20
    OFFSET %d;", $offset);
```

在实践中，要尽量避免显示出任何有关数据库的信息，尤其是数据库结构。或者，也可以选择使用数据库的存储过程和预定义指针等特性来抽象数库访问，使用户不能直接访问数据表和视图，不过这个办法又会产生其他的影响。

除此之外，在允许的情况下，使用代码或数据库系统保存查询日志也是一个好办法。显然，日志并不能防止任何攻击，但是利用查询日志可以跟踪到哪个程序曾经被尝试攻击过。

159.6 Error Reporting

错误报告对于安全加固来说是一把双刃剑，在提高安全性的同时也会有一定的害处。例如，攻击者可以通过输入不正确的数据，然后查看错误提示的类型及上下文来收集服务器的信息以便寻找弱点。

如果一个攻击者知道了一个页面所基于的表单信息，那么就可能会尝试修改变量：

Example 166 用自定义的 HTML 页面攻击变量

```
<form method="post" action="attacktarget?username=badfoo&amp;password=badfoo
    ">
<input type="hidden" name="username" value="badfoo" />
<input type="hidden" name="password" value="badfoo" />
</form>
```

通常情况下，PHP 返回的错误提示可以帮助开发者调试程序，例如错误提示中可以包含哪个文件的哪些函数或代码出错，并指出错误发生的在文件的第几行，这些就是 PHP 本身所能给出的信息。

PHP 的 show_source()、highlight_string() 或者 highlight_file() 函数在调试代码时可能会暴露出隐藏的变量、未检查的语法和其它的可能危及系统安全的信息，这种行为在运行一些具有内部调试处理的程序，或者使用通用调试技术是很危险的。如果让攻击者确定了程序是使用了哪种具体的调试技术，他们会尝试发送变量来打开调试功能：

Example 167 利用变量打开调式功能

```
<form method="post" action="attacktarget?errors=Y&amp;showerrors=1&amp;debug
    =1">
<input type="hidden" name="errors" value="Y" />
<input type="hidden" name="showerrors" value="1" />
<input type="hidden" name="debug" value="1" />
</form>
```

错误处理机制可以被攻击者用来探测系统信息，例如 PHP 的独有的错误提示风格可以说明系统在运行 PHP。

一个函数错误就可能暴露系统正在使用的数据库，或者为攻击者提供有关网页、程序或设计方面的有用信息。攻击者往往会顺藤摸瓜地找到开放的数据库端口，以及页面上某些 bug 或弱点等。比如说，攻击者可以一些不正常的数据使程序出错，来探测脚本中认证的顺序（通过错误提示的行号数字）以及脚本中其它位置可能泄露的信息。

一个文件系统或者 PHP 的错误就会暴露 Web 服务器具有的权限，以及文件在服务器上的组织结构，错误代码更可能会加剧此问题，最终导致泄漏了原本隐藏的信息。

一般情况下有三个常用的办法处理这些问题。其中，第一个是彻底地检查所有函数，并尝试弥补大多数错误，第二个是对在线系统彻底关闭错误报告，第三个是使用 PHP 自定义的错误处理函数创建自己的错误处理机制。根据不同的安全策略，三种方法可能都适用。

一个能提前阻止这个问题发生的方法就是利用 error_reporting() 来帮助使代码更安全并发现变量使用的危险之处。例如，在发布程序之前，先打开 E_ALL 测试代码来找到变量使用不当的地方，并且在准备正式发布时把 error_reporting() 的参数设为 0 来彻底关闭错误报告或者把 php.ini 中的 display_errors 设为 off 来关闭所有的错误显示以将代码隔绝于探测。

如果要迟一些再这样做，就不要忘记打开 ini 文件内的 log_errors 选项，并通过 error_log 指定用于记录错误信息的文件。

Example 168　用 E_ALL 来查找危险的变量

```
<?php
if ($username) { // Not initialized or checked before usage
    $good_login = 1;
}
if ($good_login == 1) { // If above test fails, not initialized or checked
     before usage
    readfile ("/highly/sensitive/data/index.html");
}
```

159.7　Register Globals

PHP 通过关闭配置文件中的 register_globals 来废弃了注册全局变量的机制，在 register_globals 打开时候可以把各种变量（包括来自 HTML 表单的请求变量）都注入代码。

PHP 默认在使用变量之前是无需进行初始化的，从而使得在打开 register_globals 的情况下更容易写出不安全的代码，register_globals 的关闭改变了代码内部变量和客户端发送的变量混杂在一起的糟糕情况。

Example 169　错误使用 register_globals

```
<?php
// 当用户合法的时候，赋值 $authorized = true
if (authenticated_user()) {
    $authorized = true;
}

// 由于并没有事先把 $authorized 初始化为 false,
// 当 register_globals 打开时，可能通过GET auth.php?authorized=1 来定义该变量
     值
```

```
// 所以任何人都可以绕过身份验证
if ($authorized) {
    include "/highly/sensitive/data.php";
}
```

如果 register_globals = on，那么上述的代码就很危险，关闭之后 $authorized 就不能通过 URL 请求等方式进行改变。

初始化变量是一个良好的编程习惯。例如，如果在上面的代码执行之前加入 $authorized = false，无论 register_globals 是 on 还是 off 都可以，因为用户状态被初始化为未经认证。

register_globals = on 还影响会话管理，例如 $username 也可以用在下面的代码中，而且 $username 也可能会从其它途径进来（比如说通过 URL 的 GET）。

Example 170 使用会话时同时兼容 register_globals

```
<?php
// 我们不知道 $username 的来源，但很清楚 $_SESSION 是
// 来源于会话数据
if (isset($_SESSION['username'])) {

    echo "Hello <b>{$_SESSION['username']}</b>";

} else {

    echo "Hello <b>Guest</b><br />";
    echo "Would you like to login?";

}
```

开发者采取相应的预防措施以便在伪造变量输入的时候给予警告是完全有可能的。如果事先确切知道变量是哪里来的，就可以检查所提交的数据是否是从不正当的表单提交而来。不过这不能保证变量未被伪造，还是需要攻击者去猜测应该怎样去伪造。如果不在乎请求数据来源的话，可以使用 $_REQUEST 数组，它包括了 GET、POST 和 COOKIE 的所有数据。

Example 171 探测有害变量

```
<?php
if (isset($_COOKIE['MAGIC_COOKIE'])) {
    // MAGIC_COOKIE 来自 cookie
    // 这样做是确保是来自 cookie 的数据
} elseif (isset($_GET['MAGIC_COOKIE']) || isset($_POST['MAGIC_COOKIE'])) {
```

```
  mail("admin@example.com", "Possible breakin attempt", $_SERVER['
      REMOTE_ADDR']);
  echo "Security violation, admin has been alerted.";
  exit;
} else {
  // 这一次请求中并没有设置 MAGIC_COOKIE 变量
}
```

单纯地关闭 register_globals 并不代表所有的代码都安全。每一段提交上来的数据都要进行具体的检查，永远要验证用户数据和对变量进行初始化，把 error_reporting() 设为 E_NOTICE 级别可以检查未初始化的变量。

Example 172　危险的变量用法

```
<?php
// 从用户目录中删除一个文件，或者……能删除更多的东西?
unlink ($evil_var);

// 记录用户的登陆，或者……能否在 /etc/passwd 添加数据?
fwrite ($fp, $evil_var);

// 执行一些普通的命令，或者……可以执行 rm -rf * ?
system ($evil_var);
exec ($evil_var);
```

必须时常留意自己的代码以确保每一个从客户端提交的变量都经过适当的检查，然后问自己以下一些问题：

- 此脚本是否只能影响所预期的文件？
- 非正常的数据被提交后能否产生作用？
- 此脚本能用于计划外的用途吗？
- 此脚本能否和其它脚本结合起来做坏事？
- 是否所有的事务都被充分记录了？

在写代码的时候向自己提问这些问题，否则以后可能要为了增加安全性而重写代码，虽然并不完全能保证系统的安全，但是至少可以提高安全性。

register_globals 和 magic_quotes 或者其它使编程更方便但会使某个变量的合法性，来源和其值被搞乱的设置都必须被关闭。

在开发阶段，可以使用 `error_reporting(E_ALL)` 模式来检查变量使用前是否有被检查或被初始化，这样就可以防止某些非正常的数据。

159.8 Magic Quotes

PHP 通过配置来关闭和移除了魔术引号（Magic Quote）特性，其本身是一个自动把进入 PHP 脚本的数据进行转义的过程，最好在编码时不要转义而在运行时根据需要而转义。

当打开魔术引号时，所有的 '（单引号）、"（双引号）、\（反斜线）和 NULL 字符都会被自动加上一个反斜线进行转义，和 addslashes() 作用完全相同。

PHP 提供了三个魔术引号指令：

- magic_quotes_gpc 影响到 HTTP 请求数据（GET，POST 和 COOKIE），而且不能在运行时使用 ini_set() 改变，只能在系统级别关闭。
- magic_quotes_runtime 打开时会把大部份从外部来源取得数据并返回的函数（包括从数据库和文本文件）所返回的数据使用反斜线转义。该选项可在运行的时改变，在 PHP 中的默认值为 off。
- magic_quotes_sybase 打开时会使用单引号对单引号进行转义而非反斜线。此选项会完全覆盖 magic_quotes_gpc。如果同时打开两个选项的话，单引号将会被转义成 ''，双引号、反斜线和 NULL 字符将不会进行转义。

```
; Magic quotes
;

; Magic quotes for incoming GET/POST/Cookie data.
magic_quotes_gpc = Off

; Magic quotes for runtime-generated data, e.g. data from SQL, from exec(), etc.
magic_quotes_runtime = Off

; Use Sybase-style magic quotes (escape ' with '' instead of \').
magic_quotes_sybase = Off
```

如果不能修改服务器端的配置文件，也使用.htaccess 来关闭魔术引号：

```
php_flag magic_quotes_gpc Off
```

没有理由再使用魔术引号，因为它不再是 PHP 支持的一部分，而且在处理代码时最好是更改自己的代码而不是依赖于魔术引号的开启。

魔术引号最初被引入是为了阻止 SQL 注入，不过现在已经可以使用数据库转义机制或者 prepared 语句来取代魔术引号功能。

虽然可以使用 get_magic_quotes_gpc() 来检测是否打开魔术引号，但是魔术引号的存在会影响到程序的可移植性，而且魔术引号在进行转义时会对程序的效率产生一定的影响，毕竟并不是所有的数据都需要被插入数据库中。

在运行时调用转义函数（如 addslashes()）比打开魔术引号会更有效率，因为并不是所有数据都需要被转义。例如，通过表单发送邮件时可以使用 stripslashes() 函数来处理转义。

虽然可以在运行时关闭 magic_quotes_gpc，但是这样做比较低效，适当的修改配置才是更好的办法。

Example 173　在运行时关闭魔术引号

```
<?php
if (get_magic_quotes_gpc()) {
    function stripslashes_deep($value)
    {
        $value = is_array($value) ?
                    array_map('stripslashes_deep', $value) :
                    stripslashes($value);

        return $value;
    }

    $_POST = array_map('stripslashes_deep', $_POST);
    $_GET = array_map('stripslashes_deep', $_GET);
    $_COOKIE = array_map('stripslashes_deep', $_COOKIE);
    $_REQUEST = array_map('stripslashes_deep', $_REQUEST);
}
```

Part XXXIV

Zend Engine

Chapter 160

Overview

160.1 Zend Engine1

从性能或功能方面很容易到达 PHP 的瓶颈导致了 Zend Engine1 的开发，并且以 Zend Engine1 为基础开发了 PHP4。

Zend 是语言引擎，也是 PHP 内核，PHP 是从外层展现的完整系统。

为了实现一个 web 脚本解释器，需要实现三个部分：

1. 解释器部分：分析输入代码，翻译代码，然后执行代码。
2. 功能部分：完成语言的功能（函数等）。
3. 接口部分：与 web 通信等。

Zend 完全参与第一部分，部分参与第二部分，PHP 参与第二部分和三部分，它们最终一起构成完整的 PHP 包。

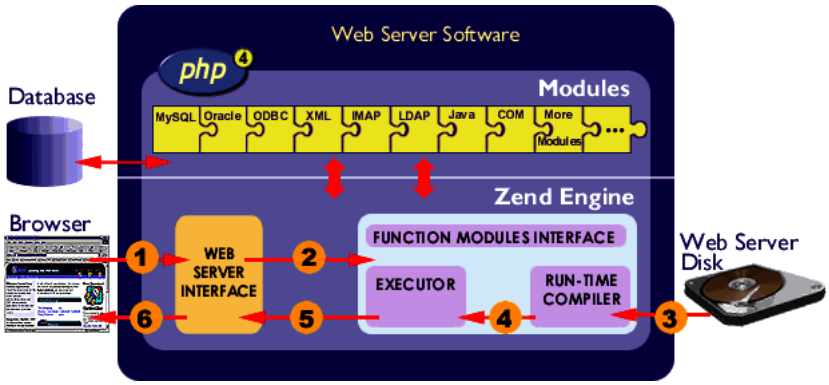

图 160.1: Web 应用程序模型

PHP 的主要组成部分包括外部扩展模块、内置模块和 Zend Engine。其中，Zend Engine

仅仅构成语言核心，并使用了预定义的函数来实现 PHP 的非常基础的部分，PHP 则包含所有的实际形成语言能力的所有模块。

160.1.1 External Modules

外部模块可以在脚本运行时使用 dl() 动态加载，具体来说就是 dl() 函数从磁盘加载一个共享的对象，并把它的功能提供给它被绑定的脚本。在脚本被终止后，外部模块将被从内存中丢弃。

外部模块不需要重新编译 PHP，这样就可以让 PHP 文件保持很小的体积并且提供足够的功能，只是外部模块需要在每次请求时都重新加载并且在请求处理结束后立即卸载。

每一个需要使用外部模块提供的函数的 PHP 脚本都必须使用 dl() 函数或者在 php.ini 中的配置项来加载外部模块。

外部模块为使用 PHP 快速开发附加功能提供了最好的结果，在频繁使用和大型项目中其提供的好处大大抵消了频繁加载和代码复杂性的不足。

外部模块可以脱离 PHP 进行额外的配置，不需要重新编译 PHP，PHP 外部模块和 PHP 主模块完全分离的特性使得可以把应用程序方便地部署在生产环境中。

160.1.2 Build-in Modules

内置模块直接编译成 PHP 并且可以绑定到每个 PHP 进程中，因此内置模块提供的功能可以立即应用到正在运行的每个 PHP 脚本中。

内置模块无需动态加载，其提供的功能即时可用，而且内置模块本身都存储在 PHP 组件库中，只要内存足够就不会产生效率问题。

内置模块的每个修改都需要重新编译 PHP，不过内置模块提供的函数库通常保持不变，这样就可以保持所有的 PHP 脚本的函数调用的可用性。

160.2 Zend Engine2

160.3 Zend Engine3

Part XXXV

SOA

Chapter 161

Overview

PHP 本身只是一个单线程的 Web 开发及系统集成的脚本开发语言。

并发请求是指将一组多个 API 调用的名称和参数分发给多个服务器同时执行待所有 api 都执行成功并返回结果后将结果整理一起返回给请求方， 当然也可以使用线程来完成，但是 PHP 的单线程特性不适合处理并发请求，需要使用 Swoole 扩展来和其他服务器进行通信。

当请求一个接口时，执行完毕后需要把结果返回给调用者后才能继续执行下一跳指令的请求就是同步的，异步请求只要把请求下发，不需要等待返回结果就可以执行下一个操作的请求是异步的，不会阻塞代码执行，不过也可能在某个时刻聚合执行结果。

PHP 处理每个请求时都需要重新加载并解析一次脚本，平时流量不多的时候看不出来，当业务发展后业务复杂度增加而 PHP 加载的文件越来越多，磁盘 IO 负载不断增加，并发请求增多导致磁盘效率下降，最终每次请求都会渐渐变慢。为了解决这个问题，可以使用 opcache 扩展缓存 PHP 解析后的代码，文件如果没有更新则只需解析一次后就一直在内存中。

PHP 从持久层获取来的数据也不会有任何内存保存（当然其他常驻内存的语言即使能做到但也不会这么做，因为会因为数据不同步导致脏数据）当流量不断增加的时候持久层一直是个单点服务器（主从后主库也是唯一单点）前端服务器可以横向扩展但是单点的持久层是不能增加的，最后导致大量的持久层拆分及相关繁琐事宜。为了解决这个问题，可以使用 Memcache 或 Redis 缓存热数据或启用其他分布式事务持久层。如果没有统一的方案，只能使用架构针对业务特有环境特性适配各种缓存方案（例如分库分表）。

PHP 不是常驻内存，导致很多特殊服务无法实现，必须依赖外部服务。例如，计数服务只能依靠 C 或 Redis 等服务实现，而且 PHP 本身的扩展对于通讯及进程类支持很弱，因此无法使用 PHP 开发分布式计算和数据挖掘等，不过现在 PHP7.0 开始支持强类型，而且可以使用 Swoole 等强化网络通信功能。

PHP 没有强制的框架规范，无法统一制定开发标准和划分模块及项目，例如在通过 Socket 请求 RPC 服务等操作时只能采用一个隔离性极强的方式去实现。

PHP 的单进程单线程模式在当处理复杂的业务的时候只能串行执行命令，所以 CPU、磁盘、内存等的利用率都很低，很多任务都需要排队等待，需要使用 pthead 等扩展才能并发地执行多任务以及返回结果。或者，可以通过隔离前后端的方式来处理复杂业务，前端可以通过消息中间件，同步或异步地调用一个或多个接口，但是 PHP 的 socket 扩展使用成本很高。

PHP 无法解决资源争抢问题，当使用持久层时往往有些关键数据是不能及时同步的，例如库存剩余和抢购等高并发业务都需要在持久层或者前端进行大量的拦截和锁定，但是实际情况是持久层压力很大的同时，很多其他无关的业务也受到影响。虽然可以使用队列，但是 PHP 支持的队列并不多，而且对全局单个数据或事务锁定的支持不完善，往往只能直接在持久层锁定或使用 Redis，Java 可以使用 zookeeper 等实现分布式支持来解决高并发问题。

PHP 不能很好的解决单个文件并发写的问题，当业务产生错误时无法及时的预警，文件无法异步写只能等待写完后才能解除锁定，虽然可以使用扩展插件来尝试解决并发写问题，并且将分布式日志集中监控管理来处理异步写操作，不过大部分语言都无法很好地解决这个问题。

PHP 无法解决文件共享问题，可以和 PHP 配合使用的分布式文件存储系统很少，往往只能先写到 Redis，再使用异步队列进行处理。

PHP 无法处理并发流量增加后的资源争抢严重的问题，MySQL 主从延迟往往会超过 30 秒，这样当 PHP 查询一个用户是否点过赞时，从库需要很长时间才能同步，这种情况下需要在前端使用 cookie 加锁，并对服务加上公共锁（setnx+expire），还可能要使用悲观锁和乐观锁。

PHP 的缓存解决方案往往只能使用 Memcached 等，但是使用 Memcached 来存储很多无用数据时会导致 Memcached 不断地在不同大小的 slab 中存储数据，这种情况需要多个 Memcached 集群并且规划数据缓冲规则，Memcached 无法缓存超过 1M 的数据。

PHP 无法解决 Memcached 脏数据问题，需要集中封装底层 DAO，一旦数据发生变更就自动更新相关 cache，如果更新频率极高则需要使用 Redis 进行存储，同时还需要 确认前端哪些部分的数据是需要设置过期时间等。

PHP 框架中模块层级太多就会导致代码很难被重复调用，而且模块划分不清晰还会造成很多数据重复获取，需要引入依赖注入或类自己本身缓存一些数据，但是也可能是过时的数据，还可能导致 PHP 内存溢出。

PHP 无法常驻内存以及对数据持久层依赖太大等问题，需要考虑使用 MySQL 代理以及 Redis 一致性 Hash 代理等尝试解决。

PHP 不支持长连接以及异步并发调用的问题可以考虑使用 Swoole、Socket、Libuv 和 Libevent 等尝试解决。

在实际的应用中用到的 Swoole 的特性包括 TCP 定长包头非定长 body 通讯协议，客户端长连接，task 同步异步处理业务逻辑并定期重启（清理未回收资源），worker 接收 task 结果并判断是否全执行完毕以此返回结果（并发处理任务），客户端超时以及 worker 进程管理。

Chapter 162

Dora-RPC

随着业务和规模的扩大，需要对很多服务和操作进行抽象和隔离，通过分层操作把服务划分为前端服务和后端服务，并且尝试通过隔离项目来管理各个项目之间的依赖关系。例如，业务层可以划分为订单服务、用户中心、权限管理和查询服务等，底层服务可以划分为分词服务、队列服务、推送服务、计数服务、短信服务和邮件通知服务等。

拆分带来的好处就是可以向平台化更进一步，其中前端承载展示界面并拼装组合来调用后端的业务和服务，后端业务逻辑不承压，但是需要处理各种复杂的业务逻辑，而且后端本身是一组或多组独立的简洁的业务逻辑封装模块，它们为前端提供了丰富的标准化的接口。

上述的架构演进其实是逐渐向 SOA 发展，而且 SOA 在抽象良好的情况下可以集成不同的功能。例如，使用前端的逻辑快速的将过去的服务接口拼装出各种所需业务或者一个业务执行了对外订阅的业务发送通报，怎么处理只需要其他业务根据需要来决定。

项目越复杂隔离性越重要，因此 SOA 实施的过程中需要架构和开发人员不断抽象总结，最终组织出标准化的后端的接口，而且 SOA 需要有很多底层服务的支撑才能完成。

有很多场景并不适合使用 SOA 来实现，例如内容发布系统只要生成静态页面，那么使用接口就可以满足，不断变化迭代的项目反而不适合使用接口。

SOA 中的依赖管理可能会很复杂，所以需要对需要对外公开的服务进行规划，并分析好现状以明确哪些项目适合，哪些不适合，因此 Java 有自己的集成消息总线及中间件和标准等。

相比 SOA，RESTful 并不适合互联网，只是简单的增删改查并不能诠释 API 的所有功能，RESTful 适合封装数据接口，但是业务接口事实上是 SOA 里面最重要的组成部分之一。

Dora RPC 是一款基础于 Swoole 定长包头通讯协议的最精简的 RPC，可以用于复杂项目前后端分离，分离后项目都通过 API 工作，可以更好的跟踪、升级、维护及管理。

Dora-RPC 旨在使用 PHP 来实现一个企业级的 SOA 业务架构，通过这个架构可以快速实现如下的功能：

- 内部 SAAS 及完善的监控管理
- 动态可伸缩式的后端
- 更简单的内部 API 集成管理
- 分布式调试支撑

在使用 SOA 来分离前后端后，用户的一个请求过来后前端往往会产生平均 10 到 200 个请求到后端以完成一个用户 一次的页面请求，因此常规的数据请求接口和业务接口需要有明确的划分，否则页面渲染会很慢还难以降低后端压力，实际运行的效率也很低，因为这些接口很有可能是串行调用，并且是逐个阻塞执行的。

业务接口权限往往权限很大，而且端口回收速度很慢，因此除了让前端直接和后端通信之外，还可以使用中间件对消息进行转发和控制，这样就可以通过检测应用服务器的返回结果来判断后端服务器是否太忙或者宕机，从而通过 PHP 代码将故障机摘除或者降低投放任务量，并将失败的请求转发到其他存活的应用服务器。

162.1 Overview

Dora-RPC 使用 Swoole 扩展进行再次封装来实现 PHP 的前后端分离，当项目变得庞大的时候需要隔离一些功能作为公共服务，Dora-RPC 希望可以将复杂的系统通过 API 服务变的简单和易维护，同时更好评估复杂项目和跨项目组调用更加清晰。

前后端分离的好处是可以减少业务抽象接口的代码依赖，并且根据日志记录统计调用频率分布，与此同时带来的不足就是通信压力增大，需要使用 Swoole 等作为 PHP 的 RPC 通信支撑。

如果一个用户请求的前端过多仍旧会很慢，需要对展示结果没有影响的处理发送到后端异步执行，执行没有先后顺序的请求并发下发到后端应用服务器并发执行等都有结果后一起返回结果，但是这也导致了持久层会承受很大压力，每个请求过来后都会并发下发 10 到 20 个并发任务给后端去处理。

如果前端发送 4000 个请求，后端就要并发 80000 个处理进程，而且还是瞬时并行的，要求后端数据服务连接数也同时增加 20 倍。例如，在使用 Swoole 实现 PHP 连接池时需要启动大量的 worker 进程，往往一个服务器就至少启动了 4000 个 task。

Dora-RPC 将服务器分为两组：前端和后端，其中前端负责展现和拼装，复杂的业务逻辑由后端负责处理。

- 前端：负责承载服务请求，对后端提供的服务进行拼装。支持同步、异步、单个、多个任务下发。
- 后端：负责提供类似 FPM 的容器常驻内存接收前端请求。
- 监视服务：负责监视后端工作状态及配置同步
- 日志服务：日志收集及统计，服务预警及日志查询。

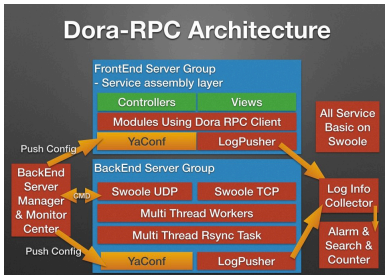

图 162.1: Dora-RPC 架构

目前 Dora-RPC 的实现只完成了基础的 RPC 功能，支持同步、异步、单个、多个任务下发，缺少的功能如下：

- 本地日志监控及推送（swoole 的 process 实现）
- 分布式日志统计及收集
- 分布式调试日志及查询
- 日志查询及索引
- 预警通知系统，支持邮件，短信，及自定义模块
- 后端前端服务器监控及界面
- 配置同步管理及推送机制
- 更好的监控规则
- 容器工作状态监控
- 异步队列

Dora-RPC 框架仅提供一套最基本的服务端和客户端，通过简单集成即可使用在任何开发框架上。

Dora-RPC 框架返回的数据结构分为三层，每层都有 msg 和 code，第一层（底层）代表当前客户端通讯情况，第二层代表当前 api 在服务端执行情况是否有异常或者投递情况，第三层代表代码执行情况，例如如果业务代码产生异常自动会记录在这里，使用这个方式是因为 RPC 本身的复杂性，通过提供这个方式可以判断系统真正的状态。

Dora-RPC 框架底层升级来支持异步任务，服务端处理完异步任务后会向客户端发送处理结果，如果当前客户端向服务端下发新任务并执行 Recive 的时候，之前客户端是无法判断出这次返回的内容是否为此次请求，与此同时由于长链接的特殊性，上一次的请求如果被异常中断没有执行 recive 的话，会导致上次的结果被本次请求拿到。为了防止以上问题，Dora-RPC 使用 guid 判断完成了异步任务结果的收集和非本次请求的返回数据的鉴别。

生产环境中的复杂的业务代码不能保证不会因为一些特殊原因产生故障，而且很多故障会导致进程终止，为此业务代码都是在 task 上执行，而不是 worker 上，这样就可以实现很多特殊的功能。

Dora-RPC 支持服务分组，很多情况下的服务是在一个公共池中，不可避免地都会受到

影响，而且不同业务部署在不同的服务器上也很常见，这些都要求 Dora-RPC 可以提供服务分组功能，规定某个服务器属于哪几个组，通过组能找到正确的服务器配置进行连接工作。

在 Dora-RPC 的任务下发模式中，下发任务数量可以分为如下：

- 并发多个任务下发
- 单个任务下发

下发任务等待结果有多个模式，例如：

- 下发任务阻塞当前进程等待处理完毕并拿到返回数据
- 下发异步任务不等待任务处理结果，只会返回下发成功，而任务在服务器跑完毕后没有任何返回
- 下发异步任务不等待任务在服务器的处理，下发成功后马上返回下发成功，而服务完成后服务端返回处理结果给客户端，客户端通过一个函数统一将所有之前异步任务结果获取返回给当前业务进程。

通过以上方式，Dora-RPC 可以成倍的增加接口响应速度和服务器利用能力，即使不使用复杂的多线程也可以达到相同的目标。

在 Dora-RPC 的网络通信中，客户端可以在 PHP-FPM 及 CLI 下工作，并且可以维持长链接。

每次来回一次请求网络耗时在 0.002 ～ 0.004 左右（也取决于内网网络），请求处理完毕后链接仍旧保留不关闭减少握手耗时及客户端队服务器的端口数消耗。

RPC 单线程客户端和虚拟机下的测试 QPS 大概在 200 ～ 500 次每秒通讯，每次下发的任务可以很多，在单应用服务器中的 QPS 可以达到 2000。

162.1.1 Gzip

若数据较大，支持包压缩 gzip 若有特殊需要可简单更改为其他压缩方式。

162.1.2 Serialize

序列化使用的 PHP 自带的 serialize，若有需要可以直接自行更换。

162.1.3 Timeout

连接和接收结果默认超时 3 秒，在 const 文件内设置，如果出现超时，客户端会自动使用其他配置进行尝试重试指定次数后返回 json 描述错误原因。如果有需要可以在重试逻辑内增加重试监控，监测各个服务器的工作状态。

162.1.4 IPFilter

如果在本次请求中有一个 ip 已经失败过一次会自动在本次内进行屏蔽，如果发现所有配置已经都尝试或超过重试次数会返回失败。cli 下失败一次后再次下发 API 时会取消掉之前所有屏蔽配置继续重试。

162.1.5 Proxy

客户端连接反向代理会消耗一定时间，因此 Dora-RPC 当前的通讯架构并没有加入反向代理服务，而且反向代理服务器虽然性能高但是本身也是一个“单点”，对于分布式服务来说还是直连方式比较方便。

对于 PHP 开发者来说，很少使用 PHP 实现大型的复杂的软件，因为开始的时候就会下意识的将项目拆分成小的组件，然后通过各种各样的 API 相互调用，以此来避免过于庞大的代码维护和跨部门调用。

如果使用模板引擎，并且在项目快速迭代时很容易出现的情况就是代码的相互依赖越来越严重。例如，一个项目可能需要集成三个到五个以上的项目提供的服务，这种情况下如果在底层更改了一个字段的名称可能就会造成大量系统不能正常工作的情况，在执行这类修改时如果修改人没有充分的“觉悟”是不会通知其他开发人员的，而且这种事情在 PHP 环境下发生之后并不是马上就能发现的（系统往往会用不存在的字段作为判断条件而导致逻辑工作方向异常。

如果要实现在开发过程中主动发现这类问题，需要将一个复杂的问题简化到一个级别后才能进行管理，这需要一个过程和切入点将一团乱麻的项目进行切片。

现实中，业务集成前端和后端的复杂度是不同的，在服务化的情况下我们可以很快的利用原有的服务能力拼装出各种丰富多样的界面和服务，而且通过服务化恰好的能够将后端的分布式复杂性和并发性与前端业务的组合复杂度完全的隔离开。

最好的效果就是即使后端底层把一个数据库被替换了或宕机了，在接口上层也不会感受到有任何变化，服务依旧稳定完整运行。通过这个方式我们还可以对我们的服务进行管理——包括功能完善、性能测试、对接口变更及依赖进行监控等以此来提高对外服务的稳定性，也可以加强对内服务的把控。

后端的复杂性的原因其实是后端服务为了一致性造成的成本是十分昂贵的，而且这不仅仅是硬件的投入，还有大量的人力消耗。例如，在互联网初期建设的时候，后端的压力承受能力有限、设备及服务昂贵，为了提高 MySQL 的查询 QPS 就需要通过大量的主从横向纵向表拆分的工作，虽然可以人工用大量的技巧解决了这个问题，但是实际上消耗还是很庞大的，而且没有人能够完全脱离业务做出一个通用的公用的可随时根据流量扩展的数据服务。

在出现了很多分布式的数据处理服务之后，但是这并不意味着后端的数据和缓存是简单容易的，而且现代的后端仍然把大量的注意力集中在“业务实施”上，很少有开发者关注一些更底层的服务。

在实际生产环境中，100 万 PV 和 20 亿 PV 的网站的区别是很大的，主要体现在服务化和自动化管理以及后端是分布式可扩展的。

在初级阶段，只是通过增加一些缓存来减轻后端数据服务压力，往往没有完善的服务治理，现实中的业务驱动则往往要求任何网站不仅仅要关注自己的业务实现，如果条件允许还要实现服务化、自动化和平台化。

事实上，服务化、自动化和平台化都是基础服务的基础支撑，只有具备了这些系统后才能提高内部的灵活性，反过来如果做的很糟糕或者很复杂，那么基础服务往往成为了累赘。

Java 本身就是一个复杂工程化的语言，因此 Java 是最早提出消息中间件、微服务、分布式、一致性概念的，对于大流量、大数据、高并发处理提供了大量开源实现，因此分布式后端服务对于 Java 来说不是很新鲜的东西。

如果仅仅关注使用 PHP 实现业务，那么在更大流量的情况下往往就需要重新调整架构或者向其他架构迁移，业务本身就已经很复杂了，很难整理和完善底层了。

如果在 PHP 框架中充斥着语法糖等过渡方案，那么对于性能的优化和企业级别的架构支撑就不可能太理想，最起码没有提供很多分布式及服务化的基础服务，不过 Zend、Symfony2、Laravel 等已经开始向这个方向开始努力，开发者也开始关注如何大流量网站的底层实现。

服务化对于大型业务流量大的互联网公司帮助很大，从最初的隔离开前后端的复杂度，让业务的实现尽量不受后端的限制以及让变更更可控等，同时这些都已经在 Java 领域的服务治理思想上有所体现。

实际上，在把业务和数据抽象成服务之后只是让业务有了一定隔离，接下来还有更多事情需要实现。例如，现实中的服务器的本地端口只有 65535 个（实际可用的还要减少 1000 个），客户端短链接每次都需要重新握手链接，然后用完后要等待超时才能回收这个端口，并发高的时候端口会不够用，虽然转换成长链接可以缓解端口限制，但是如果前端进程数超过了后端服务器的可用端口数还是会出现这类问题。

前端接口在被用户调用后往往会访问后端服务很多次，例如 100 万 PV 的前端请求传递到后端可能是 1000 万次请求，对于这类请求放大会导致后端服务器必须要比前端服务多很多，如果服务器不多就仍然不能做这类事情。

所有服务都做成 API 对外服务也并不一定可以解决上述问题，如果访问特别频繁的话，除了通过一些精简和合并（例如数据服务一次取一条改为一次取多条来减少请求次数），需要拓展的视野不仅仅在 HTTP 协议。

虽然业务开发中出现了线程模型、进程模型和协程模型等不同模型，但是如果只是写业务逻辑，还是使用单线程的方式实现是最好管理和调试的。

虽然实际上单线程 + 单进程的实现方便维护和调试，但是导致的问题就是很多可以并发的任务变成串行化了，如果使用大量的多线程来实现前端业务，对于常规的开发方式来说是一场灾难，当然可以更好的优化方式（例如 Nodejs 方式的 evenloop），不过这些方式虽然提高了 CPU 利用率，但是代价就是代码的维护成本比原先要高很多，后来才出现了 Go 的

协程、PHP 的 Yield、Swoole 的 IO 协程等方案。

单线程慢主要原因就是串行化执行代码到 IO 这些操作时阻塞导致了 CPU 空闲，Swoole 的协程细化到所有 IO 操作可以让实际开发的时候无感，虽然 Swoole 的方式是最简单的无关注的，但是只能使用它规定的驱动，PHP 的协程 Yield 实现了多个独立的代码块在人工发现有 IO 等待时调用它然后去处理其他的任务，虽然这可以实现一些灵活的调度，但不是所有人都能掌握，而且实现思路有些复杂。

在 Go 的实现中，多个并发任务的执行中分出来的“线程”有阻塞的话仍旧会阻塞等待，NodeJS 使用回调实现了完全异步操作，但是代码复杂度的提升导致调试困难。

通过以上的分析对比后发现，最好是单线程写业务、并发任务能很简单的执行而无需关注底层协层细节是对于我们来说更好的实现业务的方式。

162.2 Performance

基于 Vagrent 分配 1G 内存和 1 核 CPU 来对 Dora-RPC 进行压力测试的配置和结果如下：

- 压测进程：使用 10 个 php 进程发送请求
- 并发性能：TPS 2100 左右（比直接使用 curl 快很多）
- 响应时间：0.02 ～ 0.04s
- 后端代码为：查询一次数据库后返回结果
- CPU 使用：10% ～ 25%
- 内存使用：一个 PHP task 占用 16M 内存
- PHP 版本：5.4.41
- 压测时使用端口个数：10 个（长连接）

测试代码使用的客户端示范程序无限循环，服务端直接返回一个数组，每次接口会请求一次 api 接口调用后再下发一个请求，其中内含两个并发任务。

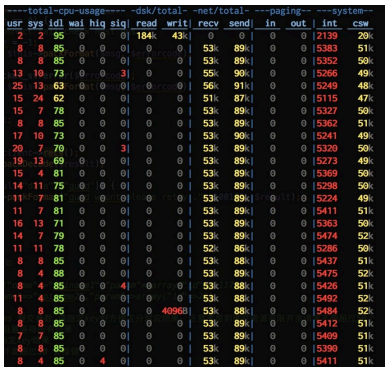

图 162.2: Dora-RPC 压力测试

- 客户端使用长链接，处理请求结束后连接也不会断开，再次使用的时候会自动找回
- 服务端自动管理 task 及进程通讯
- 通过 task 处理业务
- 如果使用更高速的序列化函数取代 serialize 会更快一些
- 支持单 api 请求，多 api 并发请求，此功能可取代发展越来越怪的 gearman
- 如果有持久化请求需求，可以考虑在此基础上自行封装下（可能导致性能下降）

另外，还可以增加中间件来检测后端服务压力状态和自动负载均衡。

162.3 Service Discovery

当服务器需要增加减少的时候，正常的流程是更新客户端的配置来实现新服务器的增加，但是这个方式有很多缺点。例如，如果某台服务器出现故障，短期内可能无法发现，发现后还是需要人工修改配置或去运维配置中心区进行摘除，通过服务发现则可以自动摘除。

增加服务器时，只要控制逻辑完善，后期可以让服务发现来把这个过程完全自动化。

Dora-RPC 提供了最简单的服务发现实现方式，使用 redis 作为存储，记录所有服务端的 ip 和端口及所在分组。

启动应用服务器时，需要通知服务发现的 redis 列表，服务器会自动将当前服务的 ip 和端口及分组上报到多个 redis 上进行注册，并定期刷新其超时时间。如果有个别节点在超过超时时间仍旧没有再上报，那么自动摘除。

客户端会定期从多个 redis 获取到最新的配置，并且更新本地配置文件内容。

162.4 Service Downgrade

当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。

服务降级方式包括服务接口拒绝服务、页面拒绝服务、延迟持久化以及随机拒绝服务等。

- 服务接口拒绝服务：无用户特定信息，页面能访问，但是添加删除提示服务器繁忙。页面内容也可在 Varnish 或 CDN 内获取。
- 页面拒绝服务：页面提示由于服务繁忙此服务暂停。跳转到 varnish 或 nginx 的一个静态页面。
- 延迟持久化：页面访问照常，但是涉及记录变更，会提示稍晚能看到结果，将数据记录到异步队列或 log，服务恢复后执行。
- 随机拒绝服务：服务接口随机拒绝服务，让用户重试。

表 162.1: 服务降级 - 持久层降级方式

数据操作动作	通过 Cache 工作	通过异步数据队列
增 insert	禁止	允许但不能有重复问题
删 delete	禁止	允许但不能有复合操作
改 update	禁止	允许只留最后结果
查 query	允许，若未命中问询 mysql 或其他持久层	走 cache

降级方式可以包括直觉管理方式和分级管理方式，其中前者要求运维人员可以指定哪些模块降级，而后者则无需运维人员关心业务细节，直接按级别降低即可。

在直觉管理方式中，当服务器检测到压力增大时，服务器监测自动发送通知给运维人员，运维人员根据自己或相关人员判断后通过配置平台设置当前运行等级来降级。

降级首先可以对非核心业务进行接口降级。如果效果不显著，开始对一些页面进行降级，以此保证核心功能的正常运行。

在分级管理方式中，当服务器检测到压力增大时，服务检测自动发送通知给运维人员，运维人员根据情况选择运行等级从 1 到 10，各个应用根据自己的级别自动判断是否工作和如何拒绝。

服务降级埋点的地方包括消息中间件、前端页面和底层数据驱动。

- 消息中间件：所有 API 调用可以使用消息中间件进行控制
- 前端页面：指定网址不可访问（NGINX+LUA）
- 底层数据驱动：拒绝所有增删改动作，只允许查询

Part XXXVI

AMQP

Chapter 163

Overview

163.1 Queue

队列（Queue）又称先进先出表（First In First Out），即先进入队列的元素，先从队列中取出。加入元素的一头叫“队头”，取出元素的一头叫“队尾”。

消息队列可以很好地异步处理数据传送和存储，如果有需要频繁地向数据库中插入数据、频繁地向搜索引擎提交数据的应用场景，就可以采取消息队列来异步插入。

另外，还可以将较慢的处理逻辑或者有并发数量限制的处理逻辑，通过消息队列放在后台处理，例如 FLV 视频转换、发送手机短信或发送电子邮件等。

163.1.1 AMQP

AMQP（Advanced Message Queuing Protocol）的目标是成为所有的消息中间件互操作的标准协议，其本质是一个功能丰富的消息队列协议。

AMQP 的最初设计者 iMatix 公司的首席执行官 Pieter Hintjens 宣布 iMatix 将退出 AMQP 工作组，而且更简单和更快的的 ZeroMQ 将不支持可能发布的 AMQP/1.0。

和 AMQP 相似的消息队列协议还有 STOMP、XMPP、MQTT 和 ActiveMQ 等，而且不同消息队列往往工作在不同的通信阶段，例如：

- AMQP 和 STOMP 和 HTTP 协议层级相同；
- MQTT 和 TCP/IP 协议的层级相同。

亚马逊 SQS(Amazon Simple Queue Service) 使用 HTTP 协议来提供消息队列，这样用户就可以使用请求/响应（request-response）语义的协议来异步地操作消息队列。

163.1.2 STOMP

STOMP（Streaming Text Oriented Messaging Protocol）是一个简单的基于文本的消息协议。

163.1.3 MQTT

MQTT（MQ Telemetry Transport）是一个专注于嵌入式设备的轻量级的消息队列协议。

163.1.4 ZeroMQ

163.2 Message Passing

UNIX 通过将链表的数组保存为消息队列来实现消息传递。每个消息队列由其在数组中的索引标识，并且具有唯一的描述符。给定的索引可以具有多个可能的描述符，UNIX 提供了访问消息传递功能的标准功能。

163.2.1 msgget()

msgget() 系统调用使用键作为参数，并返回具有匹配键的队列的描述符（如果存在）。如果它不存在，并且 IPC_CREAT 标志被设置，它使用给定的键来产生新的消息队列并返回其描述符。

163.2.2 msgrcv()

msgrcv() 用于从给定队列描述符接收消息。调用者进程必须具有队列的读取权限，而且它有两种类型：

- 如果 msgrcv() 找不到请求的消息类型就阻塞接收并将使子进程睡眠，直到另一个消息被发布在队列中，然后唤醒以再次检查。
- 非阻塞接收立即返回到调用者，并说明接收失败。

163.2.3 msgctl()

msgctl() 用于更改消息队列参数（例如所有者），而且最重要的是，它通过传递 IPC_RMID 标志来删除消息队列。

消息队列只能由其创建者，所有者或超级用户删除。

163.3 Message Queue

消息队列最初是一种进程间通信或同一进程的不同线程间的通信方式，软件队列通常用来处理一系列来自用户的输入，组通信系统可以实现和消息队列类似的功能。

消息队列非常独特，因为两个进程不必同时存在，即一个进程可以发送一个消息并退出，而该消息可以在数天后才被另一个进程获得。

消息队列本身是异步的，它允许接收者在消息发送很长时间后再取回消息，这和大多数通信协议是不同的。例如，WWW 中使用的 HTTP 协议（HTTP/2 之前）是同步的，因为客户端在发出请求后必须等待服务器回应。

很多情况下，用户需要异步的通信协议。比如，一个进程通知另一个进程发生了一个事件，但不需要等待回应，不过消息队列的异步特点也造成了一个缺点，就是接收者必须轮询消息队列才能收到最近的消息。

- 和信号相比，消息队列能够传递更多的信息；
- 与管道相比，消息队列提供了有格式的数据；
- 和信号和管道相比，消息队列仍然有大小限制。

消息队列通常都提供了异步的通信协议，每一个队列中的记录包含了详细说明的数据(例如发生的时间、输入设备的种类以及特定的输入参数等)。换句话说，消息的发送者和接收者不需要同时与消息队列互交，消息会保存在队列中，直到接收者取回它。

例如，Windows 系统通过某些形式（通常是事件的时间或是重要性的顺序）的事件队列来保存用户触发的事件，并且由系统把每个事件从事件队列中传递给目标应用程序。

在消息队列的实现中，消息队列通常保存在链表结构中并提供有格式的字节流，拥有权限的进程可以向消息队列中写入或读取消息。

消息队列通常使用发布/订阅（publisher/subscriber）模式，而且其典型应用是作为基于服务的中间件系统组成的一部分，大多数消息系统（例如 JMS(Java Message Service)）都在其提供的 API 中同时支持发布/订阅模式和消息队列。

消息队列的开源实现包括 Apache ActiveMQ、Apache Qpid、Apache Kafka、Beanstalkd、IBM MQ、JBoss Messaging、JORAM、RabbitMQ、Sun Open Message Queue、Tarantool 和 HTTPSQS 等。

消息队列除了可以作为不同线程或进程间的缓冲外，还可以通过消息队列当前消息数量来检测接收线程或进程性能是否有问题。

Chapter 164

ActiveMQ

Chapter 165

RabbitMQ

RabbitMQ 是一个实现了 AMQP 协议的开源 message broker 软件（或基于消息的中间件（message-oriented middleware)），其中 RabbitMQ 服务器是一个基于 Open Telecom Platform 框架并使用 Erlang 语言来管理集群和失效转移的实现，RabbitMQ 客户端可以使用大多数编程语言实现。

下面的示例使用 Python 通过 RabbitMQ 消息队列来发送和接收消息。

165.1 Sending

下面的示例创建一个连接并且在确认消息队列存在后就发送一条消息，然后关闭连接。

```
#! /usr/bin/env python
import pika
connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()
channel.queue_declare(queue='hello')
channel.basic_publish(exchange='',routing_key='hello',body='Hello world!')
print(" [x] Sent 'Hello world!'")
connection.close()
```

165.2 Receiving

下面的示例从消息队列中接收消息并且在屏幕上把消息打印出来。

```
#! /usr/bin/env python
import pika
connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
```

```
channel = connection.channel()
channel.queue_declare(queue='hello')
print(' [*] Wating for messages. To exit press CTRL+C')
def callback(ch,method,properties,body):
   print(" [x] Received %r" % body)
channel.basic_consume(callback,queue='hello',no_ack=True)
channel.start_consuming()
```

Chapter 166

HTTPSQS

HTTPSQS（HTTP Simple Queue Service）是一款基于 HTTP GET/POST 协议的轻量级开源简单消息队列服务，使用 Tokyo Cabinet 的 B+Tree Key/Value 数据库来做数据的持久化存储。

HTTPSQS 具有以下特征：

- 基于 HTTP GET/POST 协议，支持 HTTP 协议的编程语言均可调用；
- 执行速度非常快，入队列、出队列速度超过 10000 次/秒；
- 支持上万的高并发连接数，可以解决 C10K 问题；
- 支持多队列；
- 单个队列支持的最大队列数量最高可达 10 亿条；
- 低内存消耗，海量数据存储，存储数十 GB 的数据只需不到 100MB 的物理内存缓冲区；
- 可以在不停止服务的情况下便捷地修改单个队列的最大队列数量；
- 可以实时查看队列状态（入队列位置、出队列位置、未读队列数量、最大队列数量）；
- 可以查看指定队列 ID（队列点）的内容，包括未出、已出的队列内容；
- 查看队列内容时，支持多字符集编码；

采用 Apache ab 命令进行压力测试 HTTPSQS，开启 10 个线程，放入 10 万条文本数据（每条 512 字节）到队列中:

- 使用 HTTP Keep-Alive 时：23018 requests/sec
- 关闭 HTTP Keep-Alive 时：11840 requests/sec

采用 Apache ab 命令进行压力测试，开启 10 个线程，从队列中取出 10 万条文本数据（每条 512 字节）：

- 使用 HTTP Keep-Alive 时：25982 requests/sec
- 关闭 HTTP Keep-Alive 时：13294 requests/sec

在金山游戏官网的生产环境中，新闻、论坛帖子、客服公告、SNS 社区等发生的增、删、改操作，文本内容实时写入 HTTPSQS 队列，全站搜索引擎增量索引准实时（1 分钟内）更

新的数据源取自 HTTPSQS，而且来自支持 Web 服务器的入队列操作和来自命令行脚本的批量入、出队列操作。

1. 金山通行证（https://my.xoyo.com）
   - 队列应用类型：手机短信上行、手机短信下发、邮件下发
   - 队列应用要求：稳定性高，存储数据量大
   - 队列部署结构：一主、一备两台 HTTPSQS 热备模式
2. 金山用户行为分析系统（http://kbi.xoyo.com）
   - 队列应用类型：用户鼠标点击、访问 URL 原始数据采集
   - 队列应用要求：并发性能高，存储数据量大
   - 队列部署结构：多台 HTTPSQS 应用层哈希分布式模式
3. 金山网络游戏运营平台 KingEyes
   - 队列应用类型：用户操作日志记录
4. 金山逍遥网站内搜索
   - 队列应用类型：索引准实时更新。
5. 金山逍遥网全站通用评论系统
   - 队列应用类型：评论发表
6. 金山《剑侠情缘》电视连续剧四大角色人物选秀活动（http://zt.xoyo.com/haixuan/）
   - 队列应用类型：用户上传的照片异步裁剪、缩放处理
7. 新浪邮箱（http://mail.sina.com.cn）
   - 队列应用类型：用户登陆日志记录

```
# ulimit -SHn 65535

$ wget http://httpsqs.googlecode.com/files/libevent-2.0.12-stable.tar.gz
$ tar zxvf libevent-2.0.12-stable.tar.gz
$ cd libevent-2.0.12-stable/
$ ./configure --prefix=/usr/local/libevent-2.0.12-stable/
$ make
# make install
$ cd ../

$ wget http://httpsqs.googlecode.com/files/tokyocabinet-1.4.47.tar.gz
$ tar zxvf tokyocabinet-1.4.47.tar.gz
$ cd tokyocabinet-1.4.47/
$ ./configure --prefix=/usr/local/tokyocabinet-1.4.47/
# ./configure --enable-off64 --prefix=/usr/local/tokyocabinet-1.4.47/
$ make
# make install
```

```
$ cd ../

$ wget http://httpsqs.googlecode.com/files/httpsqs-1.7.tar.gz
$ tar zxvf httpsqs-1.7.tar.gz
$ cd httpsqs-1.7/
$ make
# make install
$ cd ../
```

在 32 位 Linux 操作系统上编译 Tokyo cabinet 时需要使用./configure --enable-off64 代替./configure，可以使数据库文件突破 2GB 的限制。

```
# httpsqs -h
-l <ip_addr> 监听的IP地址，默认值为 0.0.0.0
-p <num> 监听的TCP端口（默认值：1218）
-x <path> 数据库目录，目录不存在会自动创建（例如：/opt/httpsqs/data）
-t <second> HTTP请求的超时时间（默认值：3）
-s <second> 同步内存缓冲区内容到磁盘的间隔秒数（默认值：5）
-c <num> 内存中缓存的最大非叶子节点数（默认值：1024）
-m <size> 数据库内存缓存大小，单位：MB（默认值：100）
-i <file> 保存进程PID到文件中（默认值：/tmp/httpsqs.pid）
-a <auth> 访问HTTPSQS的验证密码（例如：mypass123）
-d 以守护进程运行
-h 显示这个帮助
```

```
$ ulimit -SHn 65535
# httpsqs -d -p 1218 -x /data0/queue
```

使用命令“killall httpsqs”、“pkill httpsqs”和“kill ‘cat /tmp/httpsqs.pid‘”来停止 httpsqs，禁止使用命令“pkill -9 httpsqs”和“kill -9 httpsqs 的进程 ID”来结束 httpsqs，否则内存中尚未保存到磁盘的数据将会丢失。

166.1 Put

入队列（将文本消息放入队列）

HTTP GET 协议（以 curl 命令为例）

```
$ curl "http://host:port/?name=your_queue_name&opt=put&data=经过URL编码的文本消息&auth=
    mypass123"
```

HTTP POST 协议（以 curl 命令为例）:

```
$ curl -d "经过URL编码的文本消息" "http://host:port/?name=your_queue_name&opt=put&auth=
    mypass123"
```

如果入队列成功，返回：

```
HTTPSQS_PUT_OK
```

如果入队列失败，返回：

```
HTTPSQS_PUT_ERROR
```

如果队列已满，返回：

```
HTTPSQS_PUT_END
```

从 HTTPSQS 1.2 版本开始，在返回给客户端的 HTTP Header 头中增加了一行“Pos: xxx”，输出当前队列的读取位置点，例如：

```
HTTP/1.1 200 OK
Content-Type: text/plain
Keep-Alive: 120
Pos: 19
Date: Thu, 18 Mar 2010 04:57:08 GMT
Content-Length: 14

HTTPSQS_PUT_OK
```

166.2 Get

出队列（从队列中取出文本消息）

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?charset=utf-8&name=your_queue_name&opt=get&auth=mypass123"
```

```
$ curl "http://host:port/?charset=gb2312&name=your_queue_name&opt=get&auth=mypass123"
```

返回消息队列的内容给客户端。

如果没有未取出的消息队列，则返回：

```
HTTPSQS_GET_END
```

从 HTTPSQS 1.2 版本开始，在返回给客户端的 HTTP Header 头中增加了一行“Pos: xxx”，输出当前队列的读取位置点，例如：

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Keep-Alive: 120
Pos: 7
Date: Thu, 18 Mar 2010 04:56:01 GMT
```

```
Content-Length: 18

消息队列内容
```

参数 charset 说明（例如：/?charset=utf-8），指定 HTTP 输出 Header 头的字符编码，即：

```
Content-Type: text/plain; charset=utf-8
```

任何在 IANA 注册的字符编码均可使用，但是并不是所有的浏览器都能解析全部的字符编码。对于中文，常用的字符编码有 utf-8、gb2312、gbk、gb18030 和 big5 等。

166.3 Status

查看队列状态（普通方式，便于浏览器查看）：

HTTP GET 协议（以 curl 命令为例）：

```
curl "http://host:port/?name=your_queue_name&opt=status&auth=mypass123"
```

返回（示例）：

```
HTTP Simple Queue Service v1.7
------------------------------
Queue Name: xoyo
Maximum number of queues: 1000000
Put position of queue (1st lap): 45
Get position of queue (1st lap): 6
Number of unread queue: 39
```

如果“队列写入点值”大于“最大队列数量值”，将重置“队列写入点”为 1，即又从 1 开始存储新的队列内容，覆盖原来队列位置点的内容：

```
HTTP Simple Queue Service v1.7
------------------------------
Queue Name: xoyo
Maximum number of queues: 1000000
Put position of queue (2st lap): 4562
Get position of queue (1st lap): 900045
Number of unread queue: 104517
```

查看队列状态（JSON 方式，便于程序处理返回内容）：

从 HTTPSQS 1.3 版本开始支持此功能。

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?name=your_queue_name&opt=status_json&auth=mypass123"
```

返回（示例）：

```
{"name":"xoyo","maxqueue":1000000,"putpos":45,"putlap":1,"getpos":6,"getlap":1,"unread"
    :39}
```

如果“队列写入点值”大于“最大队列数量值”，将重置“队列写入点”为 1，即又从 1 开始存储新的队列内容，覆盖原来队列位置点的内容：

```
{"name":"xoyo","maxqueue":1000000,"putpos":4562,"putlap":2,"getpos":900045,"getlap":1,"
    unread":104517}
```

166.4 View

查看指定队列位置点的内容：

跟一般的队列系统不同的是，HTTPSQS 可以查看指定队列 ID（队列点）的内容，包括未出、已出的队列内容。可以方便地观测进入队列的内容是否正确。

另外，假设有一个发送手机短信的队列，由客户端守护进程从队列中取出信息，并调用“短信网关接口”发送短信。但是，如果某段时间“短信网关接口”有故障，而这段时间队列位置点 300 900 的信息已经出队列，但是发送短信失败，我们还可以在位置点 300 900 被覆盖前，查看到这些位置点的内容，作相应的处理。

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?charset=utf-8&name=your_queue_name&opt=view&pos=5&auth=
    mypass123"
```

```
$ curl "http://host:port/?charset=gb2312&name=your_queue_name&opt=view&pos=19&auth=
    mypass123"
```

pos >=1 并且 <= 1000000000，返回指定队列位置点的内容。

166.5 Reset

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?name=your_queue_name&opt=reset&auth=mypass123"
```

如果重置成功，返回：

```
HTTPSQS_RESET_OK
```

如果重置失败，返回：

```
HTTPSQS_RESET_ERROR
```

更改指定队列的最大队列数量：

默认的最大队列长度（100 万条）：1000000

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?name=your_queue_name&opt=maxqueue&num=1000000000&auth=
    mypass123"
```

num >=10 并且 <= 1000000000

如果更改最大队列数量成功，则返回：

```
HTTPSQS_MAXQUEUE_OK
```

更改的最大队列数量必须大于当前的“队列写入点”。

另外，当“队列写入点”小于“队列读取点”时（即 PUT 位于圆环的第二圈，而 GET 位于圆环的第一圈时），本操作将被取消，然后返回给客户端以下信息：

```
HTTPSQS_MAXQUEUE_CANCEL
```

不停止服务的情况下，修改定时刷新内存缓冲区内容到磁盘的间隔时间：

从 HTTPSQS 1.3 版本开始支持此功能。

默认间隔时间：5 秒或 httpsqs -s <second> 参数设置的值。

HTTP GET 协议（以 curl 命令为例）：

```
$ curl "http://host:port/?name=your_queue_name&opt=synctime&num=10&auth=mypass123"
```

num >=1 and <= 1000000000，如果修改间隔时间成功，则返回：

```
HTTPSQS_SYNCTIME_OK
```

如果 num 不在 1 到 1000000000 之间，本操作将被取消，然后返回给客户端以下信息：

```
HTTPSQS_SYNCTIME_CANCEL
```

密码校验失败：

从 HTTPSQS 1.5 版本开始支持此功能。如果密码校验失败（/?auth=xxx），将返回以下信息：

```
HTTPSQS_AUTH_FAILED
```

全局错误：

如果发生全局错误（即指令、参数错误等），将返回以下信息：

```
HTTPSQS_ERROR
```

HTTPSQS 客户端代码：

```
<?php
include_once("httpsqs_client.php");
$httpsqs = new httpsqs($httpsqs_host, $httpsqs_port, $httpsqs_auth, $httpsqs_charset);

/*
1. 将文本信息放入一个队列（注意：如果要放入队列的PHP变量是一个数组，需要事先使用序列化、
    json_encode等函数转换成文本）
```

```
    如果入队列成功，返回布尔值：true
    如果入队列失败，返回布尔值：false
*/
$result = $httpsqs->put($queue_name, $queue_data);

/*
2. 从一个队列中取出文本信息
    返回该队列的内容
    如果没有未被取出的队列，则返回文本信息：HTTPSQS_GET_END
    如果发生错误，返回布尔值：false
*/
$result = $httpsqs->get($queue_name);

/*
3. 从一个队列中取出文本信息和当前队列读取点Pos
    返回数组示例：array("pos" => 7, "data" => "text message")
    如果没有未被取出的队列，则返回数组：array("pos" => 0, "data" => "HTTPSQS_GET_END")
    如果发生错误，返回布尔值：false
*/
$result = $httpsqs->gets($queue_name);

/*
4. 查看队列状态（普通方式）
*/
$result = $httpsqs->status($queue_name);

/*
5. 查看队列状态（JSON方式）
    返回示例：{"name":"queue_name","maxqueue":5000000,"putpos":130,"putlap":1,"getpos
        ":120,"getlap":1,"unread":10}
*/
$result = $httpsqs->status_json($queue_name);

/*
6. 查看指定队列位置点的内容
    返回指定队列位置点的内容。
*/
$result = $httpsqs->view($queue_name, $queue_pos);

/*
7. 重置指定队列
```

```
    如果重置队列成功，返回布尔值：true
    如果重置队列失败，返回布尔值：false
*/
$result = $httpsqs->reset($queue_name);

/*
8. 更改指定队列的最大队列数量
   如果更改成功，返回布尔值：true
   如果更改操作被取消，返回布尔值：false
*/
$result = $httpsqs->maxqueue($queue_name, $num);

/*
9. 修改定时刷新内存缓冲区内容到磁盘的间隔时间
   如果更改成功，返回布尔值：true
   如果更改操作被取消，返回布尔值：false
*/
$result = $httpsqs->synctime($num);
```

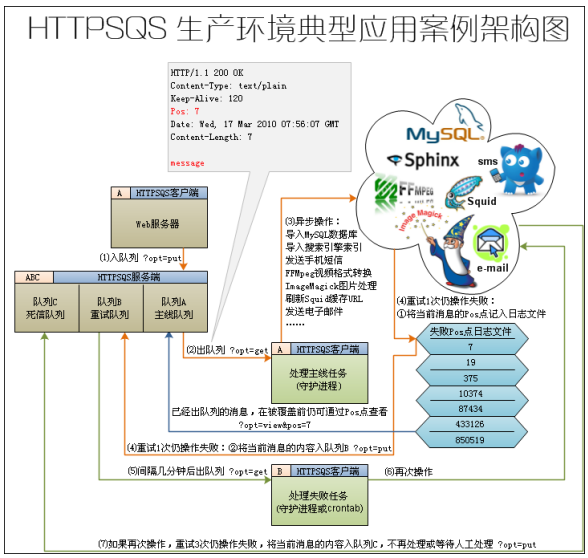

图 166.1: HTTPSQS 生产环境典型应用案例架构

一个采用 PHP 编写的 HTTPSQS 客户端简单守护进程框架如下：

假设 PHP 安装路径为/usr/local/webserver/php，使用 PHP 编写一个文件/opt/httpsqs_client_daemon.php：

```
<?php
include_once dirname(__FILE__)."/httpsqs_client.php";
```

```
$httpsqs = new httpsqs($host, $port, $auth, $charset);
while(true) {
  $result = $httpsqs->gets($name);
  $pos = $result["pos"]; //当前队列消息的读取位置点
  $data = $result["data"]; //当前队列消息的内容
  if ($data != "HTTPSQS_GET_END" && $data != "HTTPSQS_ERROR") {
    ...去做应用操作...
  } else {
    sleep(1); //暂停1秒钟后，再次循环
  }
}
```

在 Linux 下，推送到后台执行即可：

```
# nohup /usr/local/webserver/php/bin/php /opt/httpsqs_client_daemon.php 2>&1 > /dev/null
    &
```

Chapter 167

ZeroMQ

ZeroMQ 是一个为可伸缩的分布式或并发应用程序设计的高性能异步消息库。

ZeroMQ 提供一个消息队列, 但是与面向消息的中间件不同, ZeroMQ 的运行不需要专门的消息代理（message broker）, 该库提供了常见的套接字风格的 API。

ZeroMQ 类库提供一些套接字（对传统 Berkeley 套接字和 Unix domain socket 的泛化）, 每一个套接字可以代表一个端口之间的多对多连接。以消息的粒度进行操作, 套接字需要使用一种消息模式（message pattern）, 然后专门为那种模式进行了优化。

基本的 ZeroMQ 模式有:

- 请求响应模式 - 将一组客户端连接到一组服务器。这是一种远程过程调用和任务分发模式。
- 发布/订阅模式 - 将一组发布者连接到一组订阅者。这是一种数据分发模式。
- 管道模式 - 以扇出/扇入模式连接节点, 可以有多个步骤, 可以有循环。这是一种并行的任务分发和收集模式。
- 排他对模式 - 在一个排他对中连接两个套接字。（这是一种高级的为某种用例而设计的低级别模式）

任何通过套接字的消息被看作不透明的数据块。发送给订阅者的消息可以自动地通过块最开始的字符串进行过滤。ZeroMQ 提供多种消息传输协议, 包括 TCP, PGM（可靠的多播）, 进程间通信（IPC）以及线程间通讯（ITC）。

由于内部线程模型, ZeroMQ 的性能非常好, 通过利用一种自动消息批处理技术, 它甚至在吞吐量上超过了 TCP 的性能。

ZeroMQ 实现了 ZMTP, ZeroMQ 消息传输协议。

ZMTP 定义了向后兼容性的规则, 可扩展的安全机制, 命令和消息分帧, 连接元数据, 以及其他传输层功能。不使用完整的 ZeroMQ 实现, 而是直接实现 ZMTP 协议的项目不断增加。

Chapter 168

STOMP

PHP 的 Stomp 扩展可以让 PHP 应用和任何 Stomp 兼容的 Message Broker 通信，而且 Stomp 扩展支持面向对象和过程式的接口语法。

168.1 Requirement

PECL/Stomp 需要 OpenSSL、PECL/openssl 的支持才能进行 SSL 加密通信。

Part XXXVII

APC

Chapter 169

Overview

Part XXXVIII

APCu

Chapter 170

Overview

Part XXXIX

AST

Chapter 171

Overview

Part XL

BCMath

Chapter 172

Overview

涉及到金融计算的操作必须使用 BCMath 扩展，BCMath 本身是一个可以用来执行任意精度数学计算的 PHP 扩展。

对于任意精度的数学，PHP 提供了支持用字符串表示的任意大小和精度的数字的二进制计算，最多为 2147483647-1（或 0x7FFFFFFF-1）。

172.0.1 Requirement

libbcmath 和 PHP 一起发布，除此之外 BCMath 扩展不需要任何外部的库，在编译 PHP 时配置`--enable-bcmath` 就可以启用 BCMath。

```
bcmath

BCMath support => enabled

Directive => Local Value => Master Value
bcmath.scale => 0 => 0
```

172.1 Resource Type

BCMath 没有定义资源类型。

172.2 Runtime Configure

BCMath 的函数受到 php.ini 中定义的 bcmath.scale 配置指令的影响。

表 172.1: BCMath 配置选项

配置项	默认值	可修改范围
bcmath.scale	"0"	PHP_INI_ALL

172.2.1 bcmath.scale

bcmath.scale(integer) - 所有 bcmath 函数中十进制数字的数目。

172.3 Predefined Constants

BCMath 没有定义常量。

172.4 Functions

172.4.1 bcadd()

bcadd() - 2 个任意精度数字的加法计算

172.4.2 bccomp()

bccomp() - 比较两个任意精度的数字

172.4.3 bcdiv()

bcdiv() - 2 个任意精度的数字除法计算

172.4.4 bcmod()

bcmod() - 对一个任意精度数字取模

172.4.5 bcmul()

bcmul() - 2 个任意精度数字乘法计算

172.4.6 bcpow()

bcpow() - 任意精度数字的乘方

172.4.7 bcpowmod()

bcpowmod() - Raise an arbitrary precision number to another, reduced by a specified modulus

172.4.8 bcscale()

bcscale() - 设置所有 bc 数学函数的默认小数点保留位数

172.4.9 bcsqrt()

bcsqrt() - 任意精度数字的二次方根

172.4.10 bcsub()

bcsub() - 2 个任意精度数字的减法

Part XLI

Calendar

Chapter 173

Overview

Part XLII

Couchbase

Chapter 174

Overview

Part XLIII

Ctype

Chapter 175

Overview

Part XLIV

Core

Chapter 176

Overview

Core 扩展是 PHP 的核心的一部分。

在这个例子中，首先定义了一个基类和该类的子类。这个基类描述了普通的蔬菜，关于它是否可食用及其颜色。子类 Spinach 添加了烹饪的方法和另一个检查是否已烹饪的方法。

```
<?php
class Vegetable { // base class with member properties and methods
  var $edible;
  var $color;
  function Vegetable($edible,$color='green'){
    $this->edible = $edible;
    $this->color = $color;
  }

  function is_edible(){
    return $this->edible;
  }

  function what_color(){
    return $this->color;
  }
} // end of class Vegetable

class Spinach extends Vegetable { // extends base class
  var $cooked = false;
  function Spinach(){
    $this->Vegetable(true,'gree');
  }
```

```
  function cook_it(){
    $this->cooked = true;
  }

  function is_cooked(){
    return $this->cooked;
  }
} // end of class Spinach
```

接下来从这些类中实例化了两个对象，并打印了它们的信息，包括了类的继承关系，同时也定义了一些实用函数，主要为了漂亮地打印出这些变量。

```
<?php
include "Vegetable.php";

// 实用函数
function print_vars($obj) {
  foreach (get_object_vars($obj) as $prop => $val) {
    echo "\t$prop = $val\n";
  }
}

function print_methods($obj) {
  $arr = get_class_methods(get_class($obj));
  foreach ($arr as $method) {
    echo "\tfunction $method()\n";
  }
}

function class_parentage($obj, $class) {
  if (is_subclass_of($GLOBALS[$obj], $class)) {
    echo "Object $obj belongs to class " . get_class($$obj);
    echo " a subclass of $class\n";
  } else {
    echo "Object $obj does not belong to a subclass of $class\n";
  }
}

// 实例化 2 对象

$veggie = new Vegetable(true, "blue");
$leafy = new Spinach();
```

```
// 打印这些对象的信息
echo "veggie: CLASS " . get_class($veggie) . "\n";
echo "leafy: CLASS " . get_class($leafy);
echo ", PARENT " . get_parent_class($leafy) . "\n";

// 显示蔬菜的属性
echo "\nveggie: Properties\n";
print_vars($veggie);

// and leafy methods
echo "\nleafy: Methods\n";
print_methods($leafy);

echo "\nParentage:\n";
class_parentage("leafy", "Spinach");
class_parentage("leafy", "Vegetable");
```

注意，在上面的例子中，对象 $leafy 是 Spinach 的实例（Vegetable 的子类），而且在脚本的最后部分会输出以下信息：

```
[...]
Parentage:
Object leafy does not belong to a subclass of Spinach
Object leafy belongs to class spinach a subclass of Vegetable
```

176.1 Requirment

Core 扩展提供了类和对象等功能。其中，类是 PHP 内置的功能，无需其他扩展即可支持类。

176.2 Runtime Configuration

Core 扩展没有在 php.ini 中定义配置指令。

176.3 Resource Type

Core 扩展没有定义资源类型。

176.4 Predefined Constants

Core 扩展没有定义预定义常量。

Chapter 177

Functions

类/对象相关的函数是 PHP 核心的一部分，用户通过类和对象相关的函数可以获得类和对象实例的相关信息。例如，可以获取对象所属的类名，也可以获取类的成员属性和方法。

除了可以获取到对象和类的关系之外，还可以获取到它们之间的继承关系（例如对象继承自哪个类）。

177.1 __autoload()

尝试加载未定义的类

177.2 call_user_method_array()

调用一个用户方法，同时传递参数数组（已废弃）

177.3 call_user_method()

对特定对象调用用户方法（已废弃）

177.4 class_alias()

为一个类创建别名

177.5 class_exists()

检查类是否已定义

```
bool class_exists ( string $class_name [, bool $autoload = true ] )
```

检查指定的类是否已定义。

- $class_name - 类名。名字的匹配是大小写敏感的。
- $autoload - 是否默认调用 __autoload。

如果由 class_name 所指的类已经定义，此函数返回 TRUE，否则返回 FALSE。注意，不再为已定义的 interface 返回 TRUE，需要使用 interface_exists()。

```
<?php
// 使用前检查类是否存在
if (class_exists('MyClass')) {
    $myclass = new MyClass();
}
```

```
<?php
function __autoload($class) {
    include($class . '.php');

    // Check to see whether the include declared the class
    if (!class_exists($class, false)) {
        trigger_error("Unable to load class: $class", E_USER_WARNING);
    }
}

if (class_exists('MyClass')) {
    $myclass = new MyClass();
}
```

177.6 get_called_class()

后期静态绑定（"Late Static Binding"）类的名称

177.7 get_class_methods()

返回由类的方法名组成的数组

177.8 get_class_vars()

返回由类的默认属性组成的数组

177.9 get_class()

返回对象的类名

177.10 get_declared_classes()

返回由已定义类的名字所组成的数组

```
Array
(
    [0] => stdClass
    [1] => Exception
    [2] => ErrorException
    [3] => Error
    [4] => ParseError
    [5] => TypeError
    [6] => ArgumentCountError
    [7] => ArithmeticError
    [8] => DivisionByZeroError
    [9] => Closure
    [10] => Generator
    [11] => ClosedGeneratorException
    [12] => DateTime
    [13] => DateTimeImmutable
    [14] => DateTimeZone
    [15] => DateInterval
    [16] => DatePeriod
    [17] => LibXMLError
    [18] => ReflectionException
    [19] => Reflection
    [20] => ReflectionFunctionAbstract
    [21] => ReflectionFunction
    [22] => ReflectionGenerator
    [23] => ReflectionParameter
    [24] => ReflectionType
```

```
    [25] => ReflectionNamedType
    [26] => ReflectionMethod
    [27] => ReflectionClass
    [28] => ReflectionObject
    [29] => ReflectionProperty
    [30] => ReflectionClassConstant
    [31] => ReflectionExtension
    [32] => ReflectionZendExtension
    [33] => LogicException
    [34] => BadFunctionCallException
    [35] => BadMethodCallException
    [36] => DomainException
    [37] => InvalidArgumentException
    [38] => LengthException
    [39] => OutOfRangeException
    [40] => RuntimeException
    [41] => OutOfBoundsException
    [42] => OverflowException
    [43] => RangeException
    [44] => UnderflowException
    [45] => UnexpectedValueException
    [46] => RecursiveIteratorIterator
    [47] => IteratorIterator
    [48] => FilterIterator
    [49] => RecursiveFilterIterator
    [50] => CallbackFilterIterator
    [51] => RecursiveCallbackFilterIterator
    [52] => ParentIterator
    [53] => LimitIterator
    [54] => CachingIterator
    [55] => RecursiveCachingIterator
    [56] => NoRewindIterator
    [57] => AppendIterator
    [58] => InfiniteIterator
    [59] => RegexIterator
    [60] => RecursiveRegexIterator
    [61] => EmptyIterator
    [62] => RecursiveTreeIterator
    [63] => ArrayObject
    [64] => ArrayIterator
    [65] => RecursiveArrayIterator
```

```
    [66] => SplFileInfo
    [67] => DirectoryIterator
    [68] => FilesystemIterator
    [69] => RecursiveDirectoryIterator
    [70] => GlobIterator
    [71] => SplFileObject
    [72] => SplTempFileObject
    [73] => SplDoublyLinkedList
    [74] => SplQueue
    [75] => SplStack
    [76] => SplHeap
    [77] => SplMinHeap
    [78] => SplMaxHeap
    [79] => SplPriorityQueue
    [80] => SplFixedArray
    [81] => SplObjectStorage
    [82] => MultipleIterator
    [83] => SessionHandler
    [84] => __PHP_Incomplete_Class
    [85] => php_user_filter
    [86] => Directory
    [87] => AssertionError
    [88] => PDOException
    [89] => PDO
    [90] => PDOStatement
    [91] => PDORow
    [92] => AMQPConnection
    [93] => AMQPChannel
    [94] => AMQPQueue
    [95] => AMQPExchange
    [96] => AMQPEnvelope
    [97] => AMQPException
    [98] => AMQPConnectionException
    [99] => AMQPChannelException
    [100] => AMQPQueueException
    [101] => AMQPExchangeException
    [102] => APCuIterator
    [103] => APCIterator
    [104] => ast\Node
    [105] => ast\Node\Decl
    [106] => CURLFile
```

```
    [107] => DOMException
    [108] => DOMStringList
    [109] => DOMNameList
    [110] => DOMImplementationList
    [111] => DOMImplementationSource
    [112] => DOMImplementation
    [113] => DOMNode
    [114] => DOMNameSpaceNode
    [115] => DOMDocumentFragment
    [116] => DOMDocument
    [117] => DOMNodeList
    [118] => DOMNamedNodeMap
    [119] => DOMCharacterData
    [120] => DOMAttr
    [121] => DOMElement
    [122] => DOMText
    [123] => DOMComment
    [124] => DOMTypeinfo
    [125] => DOMUserDataHandler
    [126] => DOMDomError
    [127] => DOMErrorHandler
    [128] => DOMLocator
    [129] => DOMConfiguration
    [130] => DOMCdataSection
    [131] => DOMDocumentType
    [132] => DOMNotation
    [133] => DOMEntity
    [134] => DOMEntityReference
    [135] => DOMProcessingInstruction
    [136] => DOMStringExtend
    [137] => DOMXPath
    [138] => finfo
    [139] => GearmanClient
    [140] => GearmanTask
    [141] => GearmanWorker
    [142] => GearmanJob
    [143] => GearmanException
    [144] => GmagickException
    [145] => GmagickPixelException
    [146] => Gmagick
    [147] => GmagickDraw
```

```
    [148] => GmagickPixel
    [149] => GMP
    [150] => gnupg
    [151] => gnupg_keylistiterator
    [152] => Grpc\Call
    [153] => Grpc\Channel
    [154] => Grpc\Server
    [155] => Grpc\Timeval
    [156] => Grpc\ChannelCredentials
    [157] => Grpc\CallCredentials
    [158] => Grpc\ServerCredentials
    [159] => ImagickException
    [160] => ImagickDrawException
    [161] => ImagickPixelIteratorException
    [162] => ImagickPixelException
    [163] => Imagick
    [164] => ImagickDraw
    [165] => ImagickPixelIterator
    [166] => ImagickPixel
    [167] => Collator
    [168] => NumberFormatter
    [169] => Normalizer
    [170] => Locale
    [171] => MessageFormatter
    [172] => IntlDateFormatter
    [173] => ResourceBundle
    [174] => Transliterator
    [175] => IntlTimeZone
    [176] => IntlCalendar
    [177] => IntlGregorianCalendar
    [178] => Spoofchecker
    [179] => IntlException
    [180] => IntlIterator
    [181] => IntlBreakIterator
    [182] => IntlRuleBasedBreakIterator
    [183] => IntlCodePointBreakIterator
    [184] => IntlPartsIterator
    [185] => UConverter
    [186] => IntlChar
    [187] => Lua
    [188] => LuaClosure
```

```
  [189] => LuaException
  [190] => MemcachePool
  [191] => Memcache
  [192] => MongoDB\BSON\Binary
  [193] => MongoDB\BSON\Decimal128
  [194] => MongoDB\BSON\Javascript
  [195] => MongoDB\BSON\MaxKey
  [196] => MongoDB\BSON\MinKey
  [197] => MongoDB\BSON\ObjectID
  [198] => MongoDB\BSON\Regex
  [199] => MongoDB\BSON\Timestamp
  [200] => MongoDB\BSON\UTCDateTime
  [201] => MongoDB\Driver\Command
  [202] => MongoDB\Driver\Cursor
  [203] => MongoDB\Driver\CursorId
  [204] => MongoDB\Driver\Manager
  [205] => MongoDB\Driver\Query
  [206] => MongoDB\Driver\ReadConcern
  [207] => MongoDB\Driver\ReadPreference
  [208] => MongoDB\Driver\Server
  [209] => MongoDB\Driver\BulkWrite
  [210] => MongoDB\Driver\WriteConcern
  [211] => MongoDB\Driver\WriteConcernError
  [212] => MongoDB\Driver\WriteError
  [213] => MongoDB\Driver\WriteResult
  [214] => MongoDB\Driver\Exception\LogicException
  [215] => MongoDB\Driver\Exception\RuntimeException
  [216] => MongoDB\Driver\Exception\UnexpectedValueException
  [217] => MongoDB\Driver\Exception\InvalidArgumentException
  [218] => MongoDB\Driver\Exception\ConnectionException
  [219] => MongoDB\Driver\Exception\AuthenticationException
  [220] => MongoDB\Driver\Exception\SSLConnectionException
  [221] => MongoDB\Driver\Exception\WriteException
  [222] => MongoDB\Driver\Exception\BulkWriteException
  [223] => MongoDB\Driver\Exception\ExecutionTimeoutException
  [224] => MongoDB\Driver\Exception\ConnectionTimeoutException
  [225] => MessagePack
  [226] => MessagePackUnpacker
  [227] => mysqli_sql_exception
  [228] => mysqli_driver
  [229] => mysqli
```

```
    [230] => mysqli_warning
    [231] => mysqli_result
    [232] => mysqli_stmt
    [233] => OAuth
    [234] => OAuthException
    [235] => OAuthProvider
    [236] => PCS\Mgr
    [237] => PharException
    [238] => Phar
    [239] => PharData
    [240] => PharFileInfo
    [241] => php\PropertyProxy
    [242] => Redis
    [243] => RedisArray
    [244] => RedisCluster
    [245] => RedisException
    [246] => RedisClusterException
    [247] => RRDGraph
    [248] => RRDCreator
    [249] => RRDUpdater
    [250] => SeasLog
    [251] => SimpleXMLElement
    [252] => SimpleXMLIterator
    [253] => SoapClient
    [254] => SoapVar
    [255] => SoapServer
    [256] => SoapFault
    [257] => SoapParam
    [258] => SoapHeader
    [259] => SQLite3
    [260] => SQLite3Stmt
    [261] => SQLite3Result
    [262] => swoole_server
    [263] => swoole\server
    [264] => swoole_timer
    [265] => swoole\timer
    [266] => swoole_event
    [267] => swoole\event
    [268] => swoole_async
    [269] => swoole\async
    [270] => swoole_connection_iterator
```

```
    [271] => swoole\connection\iterator
    [272] => swoole_exception
    [273] => swoole\exception
    [274] => swoole_server_port
    [275] => swoole\server\port
    [276] => swoole_client
    [277] => swoole\client
    [278] => swoole_http_client
    [279] => swoole\http\client
    [280] => swoole_process
    [281] => swoole\process
    [282] => swoole_table
    [283] => swoole\table
    [284] => swoole_lock
    [285] => swoole\lock
    [286] => swoole_atomic
    [287] => swoole\atomic
    [288] => swoole_http_server
    [289] => swoole\http\server
    [290] => swoole_http_response
    [291] => swoole\http\response
    [292] => swoole_http_request
    [293] => swoole\http\request
    [294] => swoole_buffer
    [295] => swoole\buffer
    [296] => swoole_websocket_server
    [297] => swoole\websocket\server
    [298] => swoole_websocket_frame
    [299] => swoole\websocket\frame
    [300] => swoole_mysql
    [301] => swoole\mysql
    [302] => swoole_mysql_exception
    [303] => swoole\mysql\exception
    [304] => swoole_module
    [305] => swoole\module
    [306] => swoole_mmap
    [307] => swoole\mmap
    [308] => swoole_channel
    [309] => swoole\channel
    [310] => swoole_redis_server
    [311] => swoole\redis\server
```

```
    [312] => tidy
    [313] => tidyNode
    [314] => CouchbaseException
    [315] => CouchbaseMetaDoc
    [316] => CouchbaseDocumentFragment
    [317] => CouchbaseN1qlIndex
    [318] => _CouchbaseCluster
    [319] => _CouchbaseBucket
    [320] => CouchbaseMutationToken
    [321] => XMLReader
    [322] => XMLWriter
    [323] => XSLTProcessor
    [324] => Yac
    [325] => Yaconf
    [326] => Yaf_Application
    [327] => Yaf_Bootstrap_Abstract
    [328] => Yaf_Dispatcher
    [329] => Yaf_Loader
    [330] => Yaf_Request_Abstract
    [331] => Yaf_Request_Http
    [332] => Yaf_Request_Simple
    [333] => Yaf_Response_Abstract
    [334] => Yaf_Response_Http
    [335] => Yaf_Response_Cli
    [336] => Yaf_Controller_Abstract
    [337] => Yaf_Action_Abstract
    [338] => Yaf_Config_Abstract
    [339] => Yaf_Config_Ini
    [340] => Yaf_Config_Simple
    [341] => Yaf_View_Simple
    [342] => Yaf_Router
    [343] => Yaf_Route_Static
    [344] => Yaf_Route_Simple
    [345] => Yaf_Route_Supervar
    [346] => Yaf_Route_Rewrite
    [347] => Yaf_Route_Regex
    [348] => Yaf_Route_Map
    [349] => Yaf_Plugin_Abstract
    [350] => Yaf_Registry
    [351] => Yaf_Session
    [352] => Yaf_Exception
```

```
  [353] => Yaf_Exception_StartupError
  [354] => Yaf_Exception_RouterFailed
  [355] => Yaf_Exception_DispatchFailed
  [356] => Yaf_Exception_LoadFailed
  [357] => Yaf_Exception_LoadFailed_Module
  [358] => Yaf_Exception_LoadFailed_Controller
  [359] => Yaf_Exception_LoadFailed_Action
  [360] => Yaf_Exception_LoadFailed_View
  [361] => Yaf_Exception_TypeError
  [362] => Yar_Server
  [363] => Yar_Client
  [364] => Yar_Concurrent_Client
  [365] => Yar_Server_Exception
  [366] => Yar_Server_Request_Exception
  [367] => Yar_Server_Protocol_Exception
  [368] => Yar_Server_Packager_Exception
  [369] => Yar_Server_Output_Exception
  [370] => Yar_Client_Exception
  [371] => Yar_Client_Transport_Exception
  [372] => Yar_Client_Packager_Exception
  [373] => Yar_Client_Protocol_Exception
  [374] => ZipArchive
  [375] => ZMQ
  [376] => ZMQContext
  [377] => ZMQSocket
  [378] => ZMQPoll
  [379] => ZMQDevice
  [380] => ZMQException
  [381] => ZMQContextException
  [382] => ZMQSocketException
  [383] => ZMQPollException
  [384] => ZMQDeviceException
  [385] => http\Exception\RuntimeException
  [386] => http\Exception\UnexpectedValueException
  [387] => http\Exception\BadMethodCallException
  [388] => http\Exception\InvalidArgumentException
  [389] => http\Exception\BadHeaderException
  [390] => http\Exception\BadUrlException
  [391] => http\Exception\BadMessageException
  [392] => http\Exception\BadConversionException
  [393] => http\Exception\BadQueryStringException
```

```
    [394] => http\Cookie
    [395] => http\Encoding\Stream
    [396] => http\Encoding\Stream\Deflate
    [397] => http\Encoding\Stream\Inflate
    [398] => http\Encoding\Stream\Dechunk
    [399] => http\Header
    [400] => http\Header\Parser
    [401] => http\Message
    [402] => http\Message\Parser
    [403] => http\Message\Body
    [404] => http\QueryString
    [405] => http\Client
    [406] => http\Client\Request
    [407] => http\Client\Response
    [408] => http\Url
    [409] => http\Env\Url
    [410] => http\Env
    [411] => http\Env\Request
    [412] => http\Env\Response
    [413] => http\Params
    [414] => mimemessage
    [415] => Memcached
    [416] => MemcachedException
)
```

177.11 get_declared_interfaces()

返回一个数组包含所有已声明的接口

177.12 get_declared_traits()

返回所有已定义的 traits 的数组

177.13 get_object_vars()

返回由对象属性组成的关联数组

177.14 get_parent_class()

返回对象或类的父类名

177.15 interface_exists()

检查接口是否已被定义

177.16 is_a()

如果对象属于该类或该类是此对象的父类则返回 TRUE

177.17 is_subclass_of()

如果此对象是该类的子类，则返回 TRUE

177.18 method_exists()

检查类的方法是否存在

177.19 property_exists()

检查对象或类是否具有该属性

177.20 trait_exists()

检查指定的 trait 是否存在

Part XLV

CURL

Chapter 178

Overview

curl（Client URL）是一个利用 URL 语法在命令行下工作的文件传输工具，支持文件上传和下载。

cURL 还包含了用于程序开发的 libcurl（由 Daniel Stenberg 创建），libcurl 库能够连接各种服务器、使用各种协议。

libcurl 目前支持的协议有 http、https、ftp、gopher、telnet、dict、file、ldap，而且 libcurl 同时还支持 HTTPS 证书、HTTP POST、HTTP PUT、FTP 上传（也能通过 PHP 的 FTP 扩展完成）、HTTP 基于表单的上传、代理、cookies、用户名 + 密码的认证。

作为一个综合传输工具，cURL 支持的通信协议有 FTP、FTPS、HTTP、HTTPS、TFTP、SFTP、Gopher、SCP、Telnet、DICT、FILE、LDAP、LDAPS、IMAP、POP3、SMTP 和 RTSP。

使用 cURL 函数的基本思想是先使用 curl_init() 初始化 cURL 会话，接着可以通过 curl_setopt() 设置需要的全部选项，然后使用 curl_exec() 来执行会话，当执行完会话后使用 curl_close() 关闭会话。

下面是一个使用 cURL 函数获取 example.com 主页保存到文件的例子：

Example 174　使用 PHP cURL 模块获取 example.com 的主页

```
<?php
$ch = curl_init('http://www.example.com/');
$fp = fopen('example_homepage.txt','w');

curl_setopt($ch,CURLOPT_FILE,$fp);
curl_setopt($ch,CURLOPT_HEADER,0);

curl_exec($ch);
curl_close($ch);
fclose($fp);
```

需要注意的是，如果设置 CURL 使用 POST 方法发送 Form 数据，并且设置了 CURLOPT_POSTFIELDS 选项，实际的 Form 数据是以 multipart 的格式发送的。

Example 175 设置 PHP cURL 模块使用 POST 方法发送数据

```
<?php
$params=['name'=>'John', 'surname'=>'Doe', 'age'=>36];

$defaults = [
  CURLOPT_URL => 'http://myremoteservice/',
  CURLOPT_POST => true,
  CURLOPT_POSTFIELDS => $params,
];

$ch = curl_init();
curl_setopt_array($ch, ($options + $defaults));
```

上述的示例产生如下的 POST 头：

```
--------------------------fd1c4191862e3566
Content-Disposition: form-data; name="name"

Jhon
--------------------------fd1c4191862e3566
Content-Disposition: form-data; name="surnname"

Doe
--------------------------fd1c4191862e3566
Content-Disposition: form-data; name="age"

36
--------------------------fd1c4191862e3566--
```

如果使用 http_build_query() 则可以产生标准的 POST 头（例如 `name=John&surname=Doe&age=36`）。

Example 176 使用 POST 方法发送数据时使用 http_build_query()

```
<?php
$params=['name'=>'John', 'surname'=>'Doe', 'age'=>36];

$defaults = [
  CURLOPT_URL => 'http://myremoteservice/',
  CURLOPT_POST => true,
```

```
  CURLOPT_POSTFIELDS => http_build_query($params),
];

$ch = curl_init();
curl_setopt_array($ch, ($options + $defaults));
```

- 传递一个数组到 CURLOPT_POSTFIELDS，cURL 会把数据编码成 multipart/form-data；
- 传递一个 URL-encoded 字符串时，数据会被编码成 application/x-www-form-urlencoded。

178.1 Requirement

PHP 的 CURL 扩展需要安装 libcurl 包才能使用 PHP 的 cURL 函数。

要 PHP 支持 cURL，必须在编译 PHP 时加上`--with-curl[=DIR]` 选项，DIR 为包含 lib 和 include 的目录路径。

- 在 include 目录中必须有一个名为 curl，包含了 easy.h 和 curl.h 的文件夹。
- 在 lib 目录里应该有一个名为 libcurl.a 的文件。

另外，还可以配置`--with-curlwrappers` 来使 cURL 使用 URL 流。

在 Windows32 环境下使用 CURL 模块时，libeay32.dll 和 ssleay32.dll 必须位于 PATH 环境变量包含的目录下，不要用 cURL 网站上的 libcurl.dll。

178.2 Resource Type

CURL 扩展定义了 2 种资源类型：cURL 句柄和 cURL 批处理句柄。

178.3 Runtime Configure

表 178.1: cURL 配置选项

配置项	默认值	可修改范围
curl.cainfo	NULL	PHP_INI_SYSTEM

178.3.1 curl.cainfo

curl.cainfo 是 CURLOPT_CAINFO 选项的一个默认值，而且这个值必须是一个绝对路径。

178.4 Predefined Constants

下列常量由 CURL 扩展定义用于 curl_setopt()、curl_multi_setopt() 和 curl_getinfo() 函数中，且仅在此扩展编译入 PHP 或在运行时动态载入时可用。

表 178.2: cURL 预定义常量

常量名	默认值	说明
CURLOPT_AUTOREFERER	integer	
CURLOPT_COOKIESESSION	integer	
CURLOPT_DNS_USE_GLOBAL_CACHE	integer	
CURLOPT_DNS_CACHE_TIMEOUT	integer	
CURLOPT_FTP_SSL	integer	
CURLFTPSSL_TRY	integer	
CURLFTPSSL_ALL	integer	
CURLFTPSSL_CONTROL	integer	
CURLFTPSSL_NONE	integer	
CURLOPT_PRIVATE	integer	
CURLOPT_FTPSSLAUTH	integer	
CURLOPT_PORT	integer	
CURLOPT_FILE	integer	
CURLOPT_INFILE	integer	
CURLOPT_INFILESIZE	integer	
CURLOPT_URL	integer	
CURLOPT_PROXY	integer	
CURLOPT_VERBOSE	integer	
CURLOPT_HEADER	integer	
CURLOPT_HTTPHEADER	integer	
CURLOPT_NOPROGRESS	integer	
CURLOPT_NOBODY	integer	
CURLOPT_FAILONERROR	integer	
CURLOPT_UPLOAD	integer	
CURLOPT_POST	integer	
CURLOPT_FTPLISTONLY	integer	
CURLOPT_FTPAPPEND	integer	
CURLOPT_FTP_CREATE_MISSING_DIRS	integer	
CURLOPT_NETRC	integer	
CURLOPT_FOLLOWLOCATION	integer	当 open_basedir 或 safe_mode 启用时，该常量无效
CURLOPT_FTPASCII	integer	
CURLOPT_PUT	integer	
CURLOPT_MUTE	integer	
CURLOPT_USERPWD	integer	
CURLOPT_PROXYUSERPWD	integer	
CURLOPT_RANGE	integer	
CURLOPT_TIMEOUT	integer	
CURLOPT_TIMEOUT_MS	integer	

常量名	默认值	说明
CURLOPT_TCP_NODELAY	integer	
CURLOPT_POSTFIELDS	integer	
CURLOPT_PROGRESSFUNCTION	integer	
CURLOPT_REFERER	integer	
CURLOPT_USERAGENT	integer	
CURLOPT_FTPPORT	integer	
CURLOPT_FTP_USE_EPSV	integer	
CURLOPT_LOW_SPEED_LIMIT	integer	
CURLOPT_LOW_SPEED_TIME	integer	
CURLOPT_RESUME_FROM	integer	
CURLOPT_COOKIE	integer	
CURLOPT_SSLCERT	integer	
CURLOPT_SSLCERTPASSWD	integer	
CURLOPT_WRITEHEADER	integer	
CURLOPT_SSL_VERIFYHOST	integer	
CURLOPT_COOKIEFILE	integer	
CURLOPT_SSLVERSION	integer	
CURL_SSLVERSION_DEFAULT	integer	
CURL_SSLVERSION_TLSv1	integer	
CURL_SSLVERSION_SSLv2	integer	
CURL_SSLVERSION_SSLv3	integer	
CURL_SSLVERSION_TLSv1_0	integer	
CURL_SSLVERSION_TLSv1_1	integer	
CURL_SSLVERSION_TLSv1_2	integer	
CURLOPT_TIMECONDITION	integer	
CURLOPT_TIMEVALUE	integer	
CURLOPT_CUSTOMREQUEST	integer	
CURLOPT_STDERR	integer	
CURLOPT_TRANSFERTEXT	integer	
CURLOPT_RETURNTRANSFER	integer	
CURLOPT_QUOTE	integer	
CURLOPT_POSTQUOTE	integer	
CURLOPT_INTERFACE	integer	
CURLOPT_KRB4LEVEL	integer	
CURLOPT_HTTPPROXYTUNNEL	integer	
CURLOPT_FILETIME	integer	
CURLOPT_WRITEFUNCTION	integer	
CURLOPT_READFUNCTION	integer	
CURLOPT_PASSWDFUNCTION	integer	
CURLOPT_HEADERFUNCTION	integer	
CURLOPT_MAXREDIRS	integer	
CURLOPT_MAXCONNECTS	integer	
CURLOPT_CLOSEPOLICY	integer	
CURLOPT_FRESH_CONNECT	integer	

常量名	默认值	说明
CURLOPT_FORBID_REUSE	integer	
CURLOPT_RANDOM_FILE	integer	
CURLOPT_EGDSOCKET	integer	
CURLOPT_CONNECTTIMEOUT	integer	
CURLOPT_CONNECTTIMEOUT_MS	integer	
CURLOPT_SSL_VERIFYPEER	integer	
CURLOPT_CAINFO	integer	
CURLOPT_CAPATH	integer	
CURLOPT_COOKIEJAR	integer	
CURLOPT_SSL_CIPHER_LIST	integer	
CURLOPT_BINARYTRANSFER	integer	
CURLOPT_NOSIGNAL	integer	
CURLOPT_PROXYTYPE	integer	
CURLOPT_BUFFERSIZE	integer	
CURLOPT_HTTPGET	integer	
CURLOPT_HTTP_VERSION	integer	
CURLOPT_SSLKEY	integer	
CURLOPT_SSLKEYTYPE	integer	
CURLOPT_SSLKEYPASSWD	integer	
CURLOPT_SSLENGINE	integer	
CURLOPT_SSLENGINE_DEFAULT	integer	
CURLOPT_SSLCERTTYPE	integer	
CURLOPT_CRLF	integer	
CURLOPT_ENCODING	integer	
CURLOPT_PROXYPORT	integer	
CURLOPT_UNRESTRICTED_AUTH	integer	
CURLOPT_FTP_USE_EPRT	integer	
CURLOPT_HTTP200ALIASES	integer	
CURLOPT_HTTPAUTH	integer	
CURLAUTH_BASIC	integer	
CURLAUTH_DIGEST	integer	
CURLAUTH_GSSNEGOTIATE	integer	
CURLAUTH_NTLM	integer	
CURLAUTH_ANY	integer	
CURLAUTH_ANYSAFE	integer	
CURLOPT_PROXYAUTH	integer	
CURLOPT_MAX_RECV_SPEED_LARGE	integer	
CURLOPT_MAX_SEND_SPEED_LARGE	integer	
CURLOPT_HEADEROPT	integer	
CURLOPT_PROXYHEADER	integer	
CURLCLOSEPOLICY_LEAST_RECENTLY_USED	integer	
CURLCLOSEPOLICY_LEAST_TRAFFIC	integer	
CURLCLOSEPOLICY_SLOWEST	integer	
CURLCLOSEPOLICY_CALLBACK	integer	

常量名	默认值	说明
CURLCLOSEPOLICY_OLDEST	integer	
CURLINFO_PRIVATE	integer	
CURLINFO_EFFECTIVE_URL	integer	
CURLINFO_HTTP_CODE	integer	
CURLINFO_HEADER_OUT	integer	
CURLINFO_HEADER_SIZE	integer	
CURLINFO_REQUEST_SIZE	integer	
CURLINFO_TOTAL_TIME	integer	
CURLINFO_NAMELOOKUP_TIME	integer	
CURLINFO_CONNECT_TIME	integer	
CURLINFO_PRETRANSFER_TIME	integer	
CURLINFO_SIZE_UPLOAD	integer	
CURLINFO_SIZE_DOWNLOAD	integer	
CURLINFO_SPEED_DOWNLOAD	integer	
CURLINFO_SPEED_UPLOAD	integer	
CURLINFO_FILETIME	integer	
CURLINFO_SSL_VERIFYRESULT	integer	
CURLINFO_CONTENT_LENGTH_DOWNLOAD	integer	
CURLINFO_CONTENT_LENGTH_UPLOAD	integer	
CURLINFO_STARTTRANSFER_TIME	integer	
CURLINFO_CONTENT_TYPE	integer	
CURLINFO_REDIRECT_TIME	integer	
CURLINFO_REDIRECT_COUNT	integer	
CURLINFO_REDIRECT_URL	string	
CURLINFO_PRIMARY_IP	string	
CURLINFO_PRIMARY_PORT	integer	
CURLINFO_LOCAL_IP	string	
CURLINFO_LOCAL_PORT	integer	
CURL_TIMECOND_IFMODSINCE	integer	
CURL_TIMECOND_IFUNMODSINCE	integer	
CURL_TIMECOND_LASTMOD	integer	
CURL_VERSION_IPV6	integer	
CURL_VERSION_KERBEROS4	integer	
CURL_VERSION_SSL	integer	
CURL_VERSION_LIBZ	integer	
CURLVERSION_NOW	integer	
CURLE_OK	integer	
CURLE_UNSUPPORTED_PROTOCOL	integer	
CURLE_FAILED_INIT	integer	
CURLE_URL_MALFORMAT	integer	
CURLE_URL_MALFORMAT_USER	integer	
CURLE_COULDNT_RESOLVE_PROXY	integer	
CURLE_COULDNT_RESOLVE_HOST	integer	
CURLE_COULDNT_CONNECT	integer	

常量名	默认值	说明
CURLE_FTP_WEIRD_SERVER_REPLY	integer	
CURLE_FTP_ACCESS_DENIED	integer	
CURLE_FTP_USER_PASSWORD_INCORRECT	integer	
CURLE_FTP_WEIRD_PASS_REPLY	integer	
CURLE_FTP_WEIRD_USER_REPLY	integer	
CURLE_FTP_WEIRD_PASV_REPLY	integer	
CURLE_FTP_WEIRD_227_FORMAT	integer	
CURLE_FTP_CANT_GET_HOST	integer	
CURLE_FTP_CANT_RECONNECT	integer	
CURLE_FTP_COULDNT_SET_BINARY	integer	
CURLE_PARTIAL_FILE	integer	
CURLE_FTP_COULDNT_RETR_FILE	integer	
CURLE_FTP_WRITE_ERROR	integer	
CURLE_FTP_QUOTE_ERROR	integer	
CURLE_HTTP_NOT_FOUND	integer	
CURLE_WRITE_ERROR	integer	
CURLE_MALFORMAT_USER	integer	
CURLE_FTP_COULDNT_STOR_FILE	integer	
CURLE_READ_ERROR	integer	
CURLE_OUT_OF_MEMORY	integer	
CURLE_OPERATION_TIMEOUTED	integer	
CURLE_FTP_COULDNT_SET_ASCII	integer	
CURLE_FTP_PORT_FAILED	integer	
CURLE_FTP_COULDNT_USE_REST	integer	
CURLE_FTP_COULDNT_GET_SIZE	integer	
CURLE_HTTP_RANGE_ERROR	integer	
CURLE_HTTP_POST_ERROR	integer	
CURLE_SSL_CONNECT_ERROR	integer	
CURLE_FTP_BAD_DOWNLOAD_RESUME	integer	
CURLE_FILE_COULDNT_READ_FILE	integer	
CURLE_LDAP_CANNOT_BIND	integer	
CURLE_LDAP_SEARCH_FAILED	integer	
CURLE_LIBRARY_NOT_FOUND	integer	
CURLE_FUNCTION_NOT_FOUND	integer	
CURLE_ABORTED_BY_CALLBACK	integer	
CURLE_BAD_FUNCTION_ARGUMENT	integer	
CURLE_BAD_CALLING_ORDER	integer	
CURLE_HTTP_PORT_FAILED	integer	
CURLE_BAD_PASSWORD_ENTERED	integer	
CURLE_TOO_MANY_REDIRECTS	integer	
CURLE_UNKNOWN_TELNET_OPTION	integer	
CURLE_TELNET_OPTION_SYNTAX	integer	
CURLE_OBSOLETE	integer	
CURLE_SSL_PEER_CERTIFICATE	integer	

常量名	默认值	说明
CURLE_GOT_NOTHING	integer	
CURLE_SSL_ENGINE_NOTFOUND	integer	
CURLE_SSL_ENGINE_SETFAILED	integer	
CURLE_SEND_ERROR	integer	
CURLE_RECV_ERROR	integer	
CURLE_SHARE_IN_USE	integer	
CURLE_SSL_CERTPROBLEM	integer	
CURLE_SSL_CIPHER	integer	
CURLE_SSL_CACERT	integer	
CURLE_BAD_CONTENT_ENCODING	integer	
CURLE_LDAP_INVALID_URL	integer	
CURLE_FILESIZE_EXCEEDED	integer	
CURLE_FTP_SSL_FAILED	integer	
CURLE_SSH	integer	
CURLFTPAUTH_DEFAULT	integer	
CURLFTPAUTH_SSL	integer	
CURLFTPAUTH_TLS	integer	
CURLPROXY_HTTP	integer	
CURLPROXY_SOCKS4	integer	
CURLPROXY_SOCKS5	integer	
CURL_NETRC_OPTIONAL	integer	
CURL_NETRC_IGNORED	integer	
CURL_NETRC_REQUIRED	integer	
CURL_HTTP_VERSION_NONE	integer	
CURL_HTTP_VERSION_1_0	integer	
CURL_HTTP_VERSION_1_1	integer	
CURLM_CALL_MULTI_PERFORM	integer	
CURLM_OK	integer	
CURLM_BAD_HANDLE	integer	
CURLM_BAD_EASY_HANDLE	integer	
CURLM_OUT_OF_MEMORY	integer	
CURLM_INTERNAL_ERROR	integer	
CURLMSG_DONE	integer	
CURLOPT_KEYPASSWD	integer	
CURLOPT_SSH_AUTH_TYPES	integer	
CURLOPT_SSH_HOST_PUBLIC_KEY_MD5	integer	
CURLOPT_SSH_PRIVATE_KEYFILE	integer	
CURLOPT_SSH_PUBLIC_KEYFILE	integer	
CURLMOPT_PIPELINING	integer	
CURLMOPT_MAXCONNECTS	integer	
CURLSSH_AUTH_ANY	integer	
CURLSSH_AUTH_DEFAULT	integer	
CURLSSH_AUTH_HOST	integer	
CURLSSH_AUTH_KEYBOARD	integer	

常量名	默认值	说明
CURLSSH_AUTH_NONE	integer	
CURLSSH_AUTH_PASSWORD	integer	
CURLSSH_AUTH_PUBLICKEY	integer	
CURL_WRAPPERS_ENABLED	integer	在编译 PHP 时指定了`--with-curlwrappers` 时有效
CURLPAUSE_ALL	integer	
CURLPAUSE_CONT	integer	
CURLPAUSE_RECV	integer	
CURLPAUSE_RECV_CONT	integer	
CURLPAUSE_SEND	integer	
CURLPAUSE_SEND_CONT	integer	
CURLPIPE_NOTHING	integer	
CURLPIPE_HTTP1	integer	
CURLPIPE_MULTIPLEX	integer	
CURLPROXY_SOCKS4A	integer	
CURLPROXY_SOCKS5_HOSTNAME	integer	

Chapter 179

CURL Function

179.1 curl_version()

获取 cURL 版本信息

```
array curl_version ([ int $age = CURLVERSION_NOW ] )
```

返回关于 cURL 的版本信息，这个关联数组包含如下元素：

表 179.1: cURL 版本信息

Indice	值描述
version_number	cURL 24 位版本号
version	cURL 版本号，字符串形式
ssl_version_number	OpenSSL 24 位版本号
ssl_version	OpenSSL 版本号，字符串形式
libz_version	zlib 版本号，字符串形式
host	关于编译 cURL 主机的信息
age	
features	一个 CURL_VERSION_XXX 常量的位掩码
protocols	一个 cURL 支持的协议名字的数组

下面的示例将会检查当前 cURL 版本使用 curl_version() 返回的`'features'`位掩码中哪些特性是可用的。

```
<?php
// 获取cURL版本数组
$version = curl_version();
```

```
// 在cURL编译版本中使用位域来检查某些特性
$bitfields = Array(
          'CURL_VERSION_IPV6',
          'CURL_VERSION_KERBEROS4',
          'CURL_VERSION_SSL',
          'CURL_VERSION_LIBZ'
          );


foreach($bitfields as $feature){
    echo $feature . ($version['features'] & constant($feature) ? ' matches' : ' does not
        match');
    echo PHP_EOL;
}
```

179.2 curl_init()

初始化一个 cURL 会话

```
resource curl_init ([ string $url = NULL ] )
```

初始化一个新的会话并返回一个 cURL 句柄（出错返回 FALSE），可以让 curl_setopt(), curl_exec() 和 curl_close() 函数使用。

- $url - 如果提供了该参数，CURLOPT_URL 选项将会被设置成这个值，同时也可以使用 curl_setopt() 函数手动地设置这个值。

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch, CURLOPT_HEADER, 0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
```

179.3 curl_getinfo()

```
mixed curl_getinfo ( resource $ch [, int $opt = 0 ] )
```

获取一个 cURL 连接资源句柄的信息，即获取最后一次传输的相关信息。

- $ch - 由 curl_init() 返回的 cURL 句柄。
- $opt - 这个参数可能是以下常量之一：

表 179.2: cURL 的 curl_getinfo() 的可选参数意义说明

配置项	默认值	可修改范围
CURLINFO_EFFECTIVE_URL	最后一个有效的 URL 地址	
CURLINFO_HTTP_CODE	最后一个收到的 HTTP 代码	
CURLINFO_FILETIME	远程获取文档的时间，如果无法获取，则返回值为“-1”	
CURLINFO_TOTAL_TIME	最后一次传输所消耗的时间	
CURLINFO_NAMELOOKUP_TIME	名称解析所消耗的时间	
CURLINFO_CONNECT_TIME	建立连接所消耗的时间	
CURLINFO_PRETRANSFER_TIME	从建立连接到准备传输所使用的时间	
CURLINFO_STARTTRANSFER_TIME	从建立连接到传输开始所使用的时间	
CURLINFO_REDIRECT_TIME	在事务传输开始前重定向所使用的时间	
CURLINFO_SIZE_UPLOAD	上传数据量的总值	
CURLINFO_SIZE_DOWNLOAD	下载数据量的总值	
CURLINFO_SPEED_DOWNLOAD	平均下载速度	
CURLINFO_SPEED_UPLOAD	平均上传速度	
CURLINFO_HEADER_SIZE	header 部分的大小	
CURLINFO_HEADER_OUT	发送请求的字符串	
CURLINFO_REQUEST_SIZE	在 HTTP 请求中有问题的请求的大小	
CURLINFO_SSL_VERIFYRESULT	通过设置 CURLOPT_SSL_VERIFYPEER 返回的 SSL 证书验证请求的结果	
CURLINFO_CONTENT_LENGTH_DOWNLOAD	从 Content-Length: field 中读取的下载内容长度	
CURLINFO_CONTENT_LENGTH_UPLOAD	上传内容大小的说明	
CURLINFO_CONTENT_TYPE	下载内容的 Content-Type: 值，NULL 表示服务器没有发送有效的 Content-Type: header	

如果 opt 被设置，以字符串形式返回它的值，否则返回一个包含下列元素的关联数组 (它们分别对应于 opt):

- `"url"`
- `"content_type"`
- `"http_code"`
- `"header_size"`
- `"request_size"`
- `"filetime"`

- "ssl_verify_result"
- "redirect_count"
- "total_time"
- "namelookup_time"
- "connect_time"
- "pretransfer_time"
- "size_upload"
- "size_download"
- "speed_download"
- "speed_upload"
- "download_content_length"
- "upload_content_length"
- "starttransfer_time"
- "redirect_time"

Example 177 curl_getinfo()

```
<?php
// 创建一个cURL句柄
$ch = curl_init('http://www.yahoo.com/');

// 执行
curl_exec($ch);

// 检查是否有错误发生
if(!curl_errno($ch)) {
 $info = curl_getinfo($ch);
 echo 'Took ' . $info['total_time'] . ' seconds to send a request to ' .
     $info['url'];
}

// Close handle
curl_close($ch);
```

如果重用 cURL 句柄，则会保留 curl_getinfo() 函数此前收集的信息，也就是说除非统计信息在 curl_getinfo() 函数内部被覆盖，否则将返回先前的信息。

179.4 curl_setopt()

设置一个 cURL 传输选项（成功时返回 TRUE，或者在失败时返回 FALSE）

```
bool curl_setopt ( resource $ch , int $option , mixed $value )
```

为 cURL 会话句柄设置选项。

- $ch - 由 curl_init() 返回的 cURL 句柄。
- $option - 需要设置的 CURLOPT_XXX 选项。
- $value - 将设置在 option 选项上的值。

179.4.1 curl option(bool)

以下 option 参数的 value 应该被设置成 bool 类型:

表 179.3: curl option(bool)

选项	设置值	备注
CURLOPT_AUTOREFERER	TRUE 时将根据 Location: 重定向时，自动设置 header 中的 Referer: 信息	
CURLOPT_BINARYTRANSFER	TRUE 时将在启用 CURLOPT_RETURNTRANSFER 时，返回原生的（Raw）输出	(已废弃) 现在 CURLOPT_RETURNTRANSFER 总是会返回原生的（Raw）内容。
CURLOPT_COOKIESESSION	TRUE 时将开启新的一次 cookie 会话。它将强制 libcurl 忽略之前会话时存的其他 cookie。会话 cookie 是指没有过期时间，只存活在会话之中	libcurl 在默认状况下无论是否为会话，都会储存、加载所有 cookie
CURLOPT_CERTINFO	TRUE 将在安全传输时输出 SSL 证书信息到 STDERR	需要开启 CURLOPT_VERBOSE 才有效
CURLOPT_CONNECT_ONLY	TRUE 将让库执行所有需要的代理、验证、连接过程，但不传输数据	此选项用于 HTTP、SMTP 和 POP3
CURLOPT_CRLF	启用时将 Unix 的换行符转换成回车换行符	
CURLOPT_DNS_USE_GLOBAL_CACHE	TRUE 会启用一个全局的 DNS 缓存	此选项非线程安全的，默认已开启
CURLOPT_FAILONERROR	当 HTTP 状态码大于等于 400，TRUE 将将显示错误详情	默认情况下将返回页面，忽略 HTTP 代码
CURLOPT_FILETIME	TRUE 时，会尝试获取远程文档中的修改时间信息	信息可通过 curl_getinfo() 函数的 CURLINFO_FILETIME 选项获取
CURLOPT_FOLLOWLOCATION	TRUE 时将会根据服务器返回 HTTP 头中的"Location: "重定向	注意，这是递归的，"Location: "发送几次就重定向几次，除非设置了 CURLOPT_MAXREDIRS，限制最大重定向次数)
CURLOPT_FORBID_REUSE	TRUE 在完成交互以后强制明确的断开连接，不能在连接池中重用	
CURLOPT_FRESH_CONNECT	TRUE 强制获取一个新的连接，而不是缓存中的连接	
CURLOPT_FTP_USE_EPRT	TRUE 时，当 FTP 下载时使用 EPRT (和 LPRT) 命令	FALSE 时禁用 EPRT 和 LPRT，仅使用 PORT 命令
CURLOPT_FTP_USE_EPSV	TRUE 时，在 FTP 传输过程中回到 PASV 模式前，先尝试 EPSV 命令	设置为 FALSE 时禁用 EPSV
CURLOPT_FTP_CREATE_MISSING_DIRS	TRUE 时，当 ftp 操作不存在的目录时将创建它	
CURLOPT_FTPAPPEND	TRUE 为追加写入文件，而不是覆盖	

选项	设置值	备注
CURLOPT_TCP_NODELAY	TRUE 时禁用 TCP 的 Nagle 算法	就是减少网络上的小包数量
CURLOPT_FTPASCII	CURLOPT_TRANSFERTEXT 的别名	
CURLOPT_FTPLISTONLY	TRUE 时只列出 FTP 目录的名字	
CURLOPT_HEADER	启用时会将头文件的信息作为数据流输出	
CURLINFO_HEADER_OUT	TRUE 时追踪句柄的请求字符串	CURLINFO_ 的前缀是有意的
CURLOPT_HTTPGET	TRUE 时会设置 HTTP 的 method 为 GET	默认是 GET，只有 method 被修改时才需要该选项
CURLOPT_HTTPPROXYTUNNEL	TRUE 会通过指定的 HTTP 代理来传输	
CURLOPT_MUTE	TRUE 时将完全静默，无论是何 cURL 函数	（已废弃）可以使用 CURLOPT_RETURNTRANSFER 作为代替
CURLOPT_NETRC	TRUE 时，在连接建立时，访问 `~/.netrc` 文件获取用户名和密码来连接远程站点	
CURLOPT_NOBODY	TRUE 时将不输出 BODY 部分，同时 Mehtod 变成了 HEAD	修改为 FALSE 时不会变成 GET
CURLOPT_NOPROGRESS	TRUE 时关闭 cURL 的传输进度	默认自动设置此选项为 TRUE，只有为了调试才需要改变设置
CURLOPT_NOSIGNAL	TRUE 时忽略所有的 cURL 传递给 PHP 进行的信号	在 SAPI 多线程传输时此项被默认启用，所以超时选项仍能使用
CURLOPT_POST	TRUE 时会发送 POST 请求，类型为 application/x-www-form-urlencoded	这是 HTML 表单提交时最常见的一种
CURLOPT_PUT	TRUE 时允许 HTTP 发送文件	要被 PUT 的文件必须在 CURLOPT_INFILE 和 CURLOPT_INFILESIZE 中设置
CURLOPT_RETURNTRANSFER	TRUE 将 curl_exec() 获取的信息以字符串返回，而不是直接输出	
CURLOPT_SAFE_UPLOAD	TRUE 禁用 @ 前缀在 CURLOPT_POSTFIELDS 中发送文件，这意味着 @ 可以在字段中安全得使用了	可使用 CURLFile 作为上传的代替
CURLOPT_SSL_VERIFYPEER	FALSE 禁止 cURL 验证对等证书	要验证的交换证书可以在 CURLOPT_CAINFO 选项中设置，或在 CURLOPT_CAPATH 中设置证书目录
CURLOPT_TRANSFERTEXT	TRUE 对 FTP 传输使用 ASCII 模式。对于 LDAP，它检索纯文本信息而非 HTML	Windows 系统不会把 STDOUT 设置成二进制模式
CURLOPT_UNRESTRICTED_AUTH	TRUE 在使用 CURLOPT_FOLLOWLOCATION 重定向 header 中的多个 location 时继续发送用户名和密码信息，哪怕主机名已改变	
CURLOPT_UPLOAD	TRUE 准备上传	
CURLOPT_VERBOSE	TRUE 会输出所有的信息，写入到 STDERR，或在 CURLOPT_STDERR 中指定的文件	

179.4.2 curl option(integer)

以下 option 参数的 value 应该被设置成 integer 类型：

表 179.4: curl option(integer)

选项	设置值	备注
CURLOPT_BUFFERSIZE	每次读入的缓冲的尺寸	不保证每次都会完全填满这个尺寸
CURLOPT_CLOSEPOLICY	CURLCLOSEPOLICY_* 中的一个	（已废弃）
CURLOPT_CONNECTTIMEOUT	在尝试连接时等待的秒数	设置为 0 则无限等待
CURLOPT_CONNECTTIMEOUT_MS	尝试连接等待的时间，以毫秒为单位	设置为 0，则无限等待。如果 libcurl 编译时使用系统标准的名称解析器（standard system name resolver），那部分的连接仍旧使用以秒计的超时解决方案，最小超时时间还是一秒钟
CURLOPT_DNS_CACHE_TIMEOUT	设置在内存中缓存 DNS 的时间	默认为 120 秒（两分钟）
CURLOPT_FTPSSLAUTH	FTP 验证方式（启用的时候）：CURLFTPAUTH_SSL（首先尝试 SSL），CURLFTPAUTH_TLS（首先尝试 TLS）或 CURLFTPAUTH_DEFAULT (让 cURL 自己决定)	
CURLOPT_HTTP_VERSION	默认为 CURL_HTTP_VERSION_NONE（让 cURL 自己判断使用哪个版本）	CURL_HTTP_VERSION_1_0 (强制使用 HTTP/1.0) CURL_HTTP_VERSION_1_1 (强制使用 HTTP/1.1)
CURLOPT_HTTPAUTH	使用的 HTTP 验证方法。选项有 CURLAUTH_BASIC、CURLAUTH_DIGEST、CURLAUTH_GSSNEGOTIATE、CURLAUTH_NTLM、CURLAUTH_ANY 和 CURLAUTH_ANYSAFE	可以使用 | 位域 (OR) 操作符结合多个值，cURL 会让服务器选择受支持的方法，并选择最好的那个。例如，CURLAUTH_ANY 是 CURLAUTH_BASIC | CURLAUTH_DIGEST | CURLAUTH_GSSNEGOTIATE | CURLAUTH_NTLM 的别名，CURLAUTH_ANYSAFE 是 CURLAUTH_DIGEST | CURLAUTH_GSSNEGOTIATE | CURLAUTH_NTLM 的别名
CURLOPT_INFILESIZE	希望传给远程站点的文件尺寸，字节 (byte) 为单位	无法用这个选项阻止 libcurl 发送更多的数据，确切发送什么取决于 CURLOPT_READFUNCTION
CURLOPT_LOW_SPEED_LIMIT	传输速度，每秒字节（bytes）数	根据 CURLOPT_LOW_SPEED_TIME 秒数统计是否因太慢而取消传输
CURLOPT_LOW_SPEED_TIME	当传输速度小于 CURLOPT_LOW_SPEED_LIMIT 时 (bytes/sec)，PHP 会根据 CURLOPT_LOW_SPEED_TIME 来判断是否因太慢而取消传输	
CURLOPT_MAXCONNECTS	允许的最大连接数量	到达限制时，会通过 CURLOPT_CLOSEPOLICY 决定应该关闭哪些连接
CURLOPT_MAXREDIRS	指定最多的 HTTP 重定向次数	这个选项是和 CURLOPT_FOLLOWLOCATION 一起使用的
CURLOPT_PORT	用来指定连接端口	

选项	设置值	备注
CURLOPT_POSTREDIR	位掩码，1 (301 永久重定向), 2 (302 Found) 和 4 (303 See Other)	设置 CURLOPT_FOLLOWLOCATION 时，什么情况下需要再次 HTTP POST 到重定向网址
CURLOPT_PROTOCOLS	CURLPROTO_* 的位掩码。	启用时，会限制 libcurl 在传输过程中可使用哪些协议。这将允许在编译 libcurl 时支持众多协议，但是限制只用允许的子集。 默认 libcurl 将使用所有支持的协议，可用的协议选项为 CURLPROTO_HTTP、CURLPROTO_HTTPS、CURLPROTO_FTP、CURLPROTO_FTPS、CURLPROTO_SCP、CURLPROTO_SFTP、CURLPROTO_TELNET、CURLPROTO_LDAP、CURLPROTO_LDAPS、CURLPROTO_DICT、CURLPROTO_FILE、CURLPROTO_TFTP、CURLPROTO_ALL
CURLOPT_PROXYAUTH	HTTP 代理连接的验证方式，使用在 CURLOPT_HTTPAUTH 中的位掩码	当前仅仅支持 CURLAUTH_BASIC 和 CURLAUTH_NTLM
CURLOPT_PROXYPORT	代理服务器的端口	端口也可以在 CURLOPT_PROXY 中设置
CURLOPT_PROXYTYPE	可以是 CURLPROXY_HTTP (默认值)、CURLPROXY_SOCKS4、CURLPROXY_SOCKS5、CURLPROXY_SOCKS4A 或 CURLPROXY_SOCKS5_HOSTNAME	
CURLOPT_REDIR_PROTOCOLS	CURLPROTO_* 值的位掩码。如果被启用，位掩码会限制 libcurl 在 CURLOPT_FOLLOWLOCATION 开启时使用的协议	默认允许除 FILE 和 SCP 外所有协议，这和 7.19.4 前的版本无条件支持所有支持的协议不同
CURLOPT_RESUME_FROM	在恢复传输时，传递字节为单位的偏移量（用来断点续传）	
CURLOPT_SSL_VERIFYHOST	设置为 1 是检查服务器 SSL 证书中是否存在一个公用名 (common name)	公用名 (Common Name) 一般来讲就是填写你将要申请 SSL 证书的域名 (domain) 或子域名 (sub domain)。 设置成 2，会检查公用名是否存在，并且是否与提供的主机名匹配。在生产环境中，这个值应该是 2（默认值）

选项	设置值	备注
CURLOPT_SSLVERSION	CURL_SSLVERSION_DEFAULT (0), CURL_SSLVERSION_TLSv1 (1), CURL_SSLVERSION_SSLv2 (2), CURL_SSLVERSION_SSLv3 (3), CURL_SSLVERSION_TLSv1_0 (4), CURL_SSLVERSION_TLSv1_1 (5), CURL_SSLVERSION_TLSv1_2 (6) 中的其中一个	最好别设置这个值，让它使用默认值。设置为 2 或 3 比较危险，在 SSLv2 和 SSLv3 中有弱点存在
CURLOPT_TIMECONDITION	默认为 CURL_TIMECOND_IFMODSINCE，设置如何对待 CURLOPT_TIMEVALUE	使用 CURL_TIMECOND_IFMODSINCE，仅在页面 CURLOPT_TIMEVALUE 之后修改，才返回页面。没有修改则返回”304 Not Modified” 头，假设设置了 CURLOPT_HEADER 为 TRUE。CURL_TIMECOND_IFUNMODSINCE 则起相反的效果
CURLOPT_TIMEOUT	允许 cURL 函数执行的最长秒数	
CURLOPT_TIMEOUT_MS	设置 cURL 允许执行的最长毫秒数。	如果 libcurl 编译时使用系统标准的名称解析器（standard system name resolver），那部分的连接仍旧使用以秒计的超时解决方案，最小超时时间还是一秒钟
CURLOPT_TIMEVALUE	秒数，从 1970 年 1 月 1 日开始	这个时间会被 CURLOPT_TIMECONDITION 使。默认使用 CURL_TIMECOND_IFMODSINCE
CURLOPT_MAX_RECV_SPEED_LARGE	If a download exceeds this speed (counted in bytes per second) on cumulative average during the transfer, the transfer will pause to keep the average rate less than or equal to the parameter value. Defaults to unlimited speed.	
CURLOPT_MAX_SEND_SPEED_LARGE	If an upload exceeds this speed (counted in bytes per second) on cumulative average during the transfer, the transfer will pause to keep the average rate less than or equal to the parameter value. Defaults to unlimited speed.	
CURLOPT_SSH_AUTH_TYPES	A bitmask consisting of one or more of CURLSSH_AUTH_PUBLICKEY, CURLSSH_AUTH_PASSWORD, CURLSSH_AUTH_HOST, CURLSSH_AUTH_KEYBOARD	Set to CURLSSH_AUTH_ANY to let libcurl pick one.

选项	设置值	备注
CURLOPT_IPRESOLVE	Allows an application to select what kind of IP addresses to use when resolving host names. This is only interesting when using host names that resolve addresses using more than one version of IP, possible values are CURL_IPRESOLVE_WHATEVER, CURL_IPRESOLVE_V4, CURL_IPRESOLVE_V6, by default CURL_IPRESOLVE_WHATEVER.	

179.4.3 curl option(string)

以下 option 参数的 value 应该被设置成 string 类型:

表 179.5: curl option(string)

选项	设置值	备注
CURLOPT_CAINFO	一个保存着 1 个或多个用来让服务端验证的证书的文件名（可能需要绝对路径）	这个参数仅仅在和 CURLOPT_SSL_VERIFYPEER 一起使用时才有意义
CURLOPT_CAPATH	一个保存着多个 CA 证书的目录	这个选项是和 CURLOPT_SSL_VERIFYPEER 一起使用的
CURLOPT_COOKIE	设定 HTTP 请求中`"Cookie: "`部分的内容。	多个 cookie 用分号分隔，分号后带一个空格 (例如`"fruit=apple; colour=red"`)
CURLOPT_COOKIEFILE	包含 cookie 数据的文件名，cookie 文件的格式可以是 Netscape 格式，或者只是纯 HTTP 头部风格，存入文件	如果文件名是空的，不会加载 cookie，但 cookie 的处理仍旧启用
CURLOPT_COOKIEJAR	连接结束后（比如调用 curl_close）保存 cookie 信息的文件	
CURLOPT_CUSTOMREQUEST	HTTP 请求时，使用自定义的 Method 来代替`"GET"` 或`"HEAD"`。对`"DELETE"` 或者其他更隐蔽的 HTTP 请求有用。有效值如 `"GET"`，`"POST"`，`"CONNECT"` 等，也就是说不要在这里输入整行 HTTP 请求，例如输入`"GET /index.html HTTP/1.0\r\n\r\n"` 是不正确的	不确定服务器支持这个自定义方法则不要使用它
CURLOPT_EGDSOCKET	类似 CURLOPT_RANDOM_FILE，除了一个 Entropy Gathering Daemon 套接字	
CURLOPT_ENCODING	HTTP 请求头中`"Accept-Encoding: "` 的值，能够解码响应的内容，支持的编码有`"identity"`，`"deflate"` 和`"gzip"`	如果为空字符串`""`，会发送所有支持的编码类型

选项	设置值	备注
CURLOPT_FTPPORT	这个值将被用来获取供 FTP 的"PORT" 指令所需要的 IP 地址，"PORT" 指令告诉远程服务器连接到我们指定的 IP 地址	这个字符串可以是纯文本的 IP 地址、主机名、一个网络接口名（UNIX 下）或者只是一个'-' 来使用默认的 IP 地址
CURLOPT_INTERFACE	发送的网络接口（interface）	可以是一个接口名、IP 地址或者是一个主机名
CURLOPT_KEYPASSWD	使用 CURLOPT_SSLKEY 或 CURLOPT_SSH_PRIVATE_KEYFILE 私钥时候的密码	
CURLOPT_KRB4LEVEL	KRB4 (Kerberos 4) 安全级别。下面的任何值都是有效的 (从低到高的顺序)："clear"、"safe"、"confidential"、"private"。如果字符串以上这些，将使用"private"	这个选项设置为 NULL 时将禁用 KRB4 安全认证，目前 KRB4 安全认证只能用于 FTP 传输
CURLOPT_POSTFIELDS	全部数据使用 HTTP 协议中的"POST" 操作来发送，如果要发送文件，在文件名前面加上 @ 前缀并使用完整路径，文件类型可以在文件名后以';type=mimetype' 的格式指定。这个参数可以是 urlencoded 后的字符串，类似'para1=val1&para2=val2&...', 也可以使用一个以字段名为键值, 字段数据为值的数组。如果 value 是一个数组, Content-Type 头将会被设置成 multipart/form-data	使用 @ 前缀传递文件时, value 必须是个数组, 现在文件可通过 CURLFile 发送。设置 CURLOPT_SAFE_UPLOAD 为 TRUE 可禁用 @ 前缀发送文件以增加安全性
CURLOPT_PRIVATE	Any data that should be associated with this cURL handle. This data can subsequently be retrieved with the CURLINFO_PRIVATE option of curl_getinfo()	cURL does nothing with this data. When using a cURL multi handle, this private data is typically a unique key to identify a standard cURL handle
CURLOPT_PROXY	HTTP 代理通道	
CURLOPT_PROXYUSERPWD	一个用来连接到代理的"[username]:[password]" 格式的字符串	
CURLOPT_RANDOM_FILE	一个被用来生成 SSL 随机数种子的文件名	
CURLOPT_RANGE	以"X-Y" 的形式，其中 X 和 Y 都是可选项获取数据的范围，以字节计	HTTP 传输线程也支持在重复项中间用逗号分隔（例如"X-Y,N-M"）
CURLOPT_REFERER	在 HTTP 请求头中"Referer: " 的内容	
CURLOPT_SSH_HOST_PUBLIC_KEY_MD5	包含 32 位长的 16 进制数值。这个字符串应该是远程主机公钥（public key）的 MD5 校验值。在不匹配的时候 libcurl 会拒绝连接	此选项仅用于 SCP 和 SFTP 的传输
CURLOPT_SSH_PUBLIC_KEYFILE	The file name for your public key. If not used, libcurl defaults to $HOME/.ssh/id_dsa.pub if the HOME environment variable is set, and just “id_dsa.pub” in the current directory if HOME is not set	

选项	设置值	备注
CURLOPT_SSH_PRIVATE_KEYFILE	The file name for your private key. If not used, libcurl defaults to $HOME/.ssh/id_dsa if the HOME environment variable is set, and just “id_dsa” in the current directory if HOME is not set. If the file is password-protected, set the password with CURLOPT_KEYPASSWD	
CURLOPT_SSL_CIPHER_LIST	一个 SSL 的加密算法列表	RC4-SHA 和 TLSv1 都是可用的加密列表
CURLOPT_SSLCERT	一个包含 PEM 格式证书的文件名	
CURLOPT_SSLCERTPASSWD	使用 CURLOPT_SSLCERT 证书需要的密码	
CURLOPT_SSLCERTTYPE	证书的类型。支持的格式有 “PEM” (默认值), “DER” 和 “ENG”	
CURLOPT_SSLENGINE	用来在 CURLOPT_SSLKEY 中指定的 SSL 私钥的加密引擎变量	
CURLOPT_SSLENGINE_DEFAULT	用来做非对称加密操作的变量	
CURLOPT_SSLKEY	包含 SSL 私钥的文件名	
CURLOPT_SSLKEYPASSWD	在 CURLOPT_SSLKEY 中指定了的 SSL 私钥的密码	这个选项包含了敏感的密码信息，需要保证这个 PHP 脚本的安全
CURLOPT_SSLKEYTYPE	CURLOPT_SSLKEY 中规定的私钥的加密类型	支持的密钥类型为 “PEM”(默认值)、“DER” 和 “ENG”
CURLOPT_URL	需要获取的 URL 地址	也可以在 curl_init() 初始化会话时设置
CURLOPT_USERAGENT	在 HTTP 请求中包含一个 “User-Agent: ” 头的字符串	
CURLOPT_USERPWD	传递一个连接中需要的用户名和密码，格式为 “[username]:[password]”	

179.4.4 curl option(array)

以下 option 参数的 value 应该被设置成 array 类型：

表 179.6: curl option(array)

选项	设置值	备注
CURLOPT_HTTP200ALIASES	HTTP 200 响应码数组	数组中的响应码被认为是正确的响应，而非错误
CURLOPT_HTTPHEADER	设置 HTTP 头字段的数组	例 如 `array('Content-type: text/plain', 'Content-length: 100')`
CURLOPT_POSTQUOTE	在 FTP 请求执行完成后，在服务器上执行的一组 array 格式的 FTP 命令	
CURLOPT_QUOTE	一组先于 FTP 请求的在服务器上执行的 FTP 命令	

179.4.5 curl option(stream)

以下 option 参数的 value 应该被设置成 stream 类型（例如使用 fopen()）：

表 179.7: curl option(stream)

选项	设置值	备注
CURLOPT_FILE	设置输出文件	默认为 STDOUT (浏览器)
CURLOPT_INFILE	上传文件时需要读取的文件	
CURLOPT_STDERR	错误输出的地址，取代默认的 STDERR	
CURLOPT_WRITEHEADER	设置 header 部分内容的写入的文件地址	

179.4.6 curl option(function)

以下 option 参数的 value 应该被设置成有效的函数或闭包类型：

表 179.8: curl option(function)

选项	设置值	备注
CURLOPT_HEADERFUNCTION	设置一个回调函数，这个函数有两个参数，第一个是 cURL 的资源句柄，第二个是输出的 header 数据	header 数据的输出必须依赖这个函数，返回已写入的数据大小
CURLOPT_PASSWDFUNCTION	设置一个回调函数，有三个参数，第一个是 cURL 的资源句柄，第二个是一个密码提示符，第三个参数是密码长度允许的最大值	返回密码的值
CURLOPT_PROGRESSFUNCTION	设置一个回调函数，有五个参数，第一个是 cURL 的资源句柄，第二个是预计要下载的总字节 (bytes) 数。第三个是目前下载的字节数，第四个是预计传输中总上传字节数，第五个是目前上传的字节数	只有设置 CURLOPT_NOPROGRESS 选项为 FALSE 时才会调用这个回调函数，返回非零值将中断传输，传输将设置 CURLE_ABORTED_BY_CALLBACK 错误
CURLOPT_READFUNCTION	回调函数名。该函数应接受三个参数。第一个是 cURL resource，第二个是通过选项 CURLOPT_INFILE 传给 cURL 的 stream resource，第三个参数是最大可以读取的数据的数量	回调函数必须返回一个字符串，长度小于或等于请求的数据量（第三个参数）。一般从传入的 stream resource 读取。返回空字符串作为 EOF（文件结束）信号
CURLOPT_WRITEFUNCTION	回调函数名。该函数应接受两个参数。第一个是 cURL resource，第二个是要写入的数据字符串	数据必须在函数中被保存，函数必须准确返回写入数据的字节数，否则传输会被一个错误所中断

179.4.7 curl option(resource)

以下 option 参数的 value 应该被设置成其他值类型：

表 179.9: curl option(resource)

选项	设置值	备注
CURLOPT_SHARE	curl_share_init() 返回的结果	使 cURL 可以处理共享句柄里的数据

Example 178 设置 cURL 选项示例

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch, CURLOPT_HEADER, false);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

//关闭cURL资源，并且释放系统资源
curl_close($ch);
```

Example 179 上传文件时的 cURL 选项示例

```
<?php
/*
  * http://localhost/upload.php:
  * print_r($_POST);
  * print_r($_FILES);
  */
$ch = curl_init();

$data = array('name' => 'Foo', 'file' => '@/home/user/test.png');

curl_setopt($ch, CURLOPT_URL, 'http://localhost/upload.php');
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_SAFE_UPLOAD, false); // PHP 5.6.0 后必须开启
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);

curl_exec($ch);
```

以上例程会输出：

```
Array
(
    [name] => Foo
)
Array
(
```

```
    [file] => Array
        (
            [name] => test.png
            [type] => image/png
            [tmp_name] => /tmp/phpcpjNeQ
            [error] => 0
            [size] => 279
        )
)
```

179.5 curl_setopt_array()

为 cURL 传输会话批量设置选项，如果全部的选项都被成功设置则返回 TRUE，只要有一个选项不能被成功设置，马上返回 FALSE，忽略其后的任何在 options 数组中的选项。

```
bool curl_setopt_array ( resource $ch , array $options )
```

curl_setopt_array() 为 cURL 传输会话批量设置选项。这个函数对于需要设置大量的 cURL 选项是非常有用的，不需要重复地调用 curl_setopt()。

- $ch - 由 curl_init() 返回的 cURL 句柄。
- $options - 一个 array 用来确定将被设置的选项及其值。数组的键值必须是一个有效的 curl_setopt() 常量或者是它们对等的整数值。

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相应的选项
$options = array(
   CURLOPT_URL => 'http://www.example.com/',
   CURLOPT_HEADER => false
);

curl_setopt_array($ch, $options);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
```

下面是一个对 curl_setopt_array() 的等价实现。

```php
<?php
if (!function_exists('curl_setopt_array')) {
   function curl_setopt_array(&$ch, $curl_options) {
       foreach ($curl_options as $option => $value) {
           if (!curl_setopt($ch, $option, $value)) {
               return false;
           }
       }
       return true;
   }
}
```

注意，对于 curl_setopt() 来说：

- 传递一个数组到 CURLOPT_POST 将会把数据以 multipart/form-data 的方式编码。
- 传递一个 URL-encoded 字符串到 CURLOPT_POST 将会以 application/x-www-form-urlencoded 的方式对数据进行编码。

179.6 curl_escape()

根据 RFC 3986 规范编码给定的字符串，返回编码后的字符串或者在失败时返回 FALSE。

```
string curl_escape ( resource $ch , string $str )
```

- $ch - 由 curl_init() 返回的 cURL 句柄。
- $str - 由 curl_init() 返回的 cURL 句柄。

Example 180 curl_escape()

```php
<?php
// 创建一个 curl 句柄
$ch = curl_init();

// 把编码后的字符串当做一个 GET 参数
$location = curl_escape($ch, 'Hofbräuhaus / München');
// 结果： Hofbr%C3%A4uhaus%20%2F%20M%C3%BCnchen

// 用编码好的字符串组装一个 URL
$url = "http://example.com/add_location.php?location={$location}";
// 结果： http://example.com/add_location.php?location=Hofbr%C3%A4uhaus%20%2F
    %20M%C3%BCnchen

// 发送 HTTP 请求并关闭句柄
```

```
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_exec($ch);
curl_close($ch);
```

179.7 curl_unescape()

解码给定的 URL 编码的字符串（返回解码后的字符串或者在失败时返回 FALSE）

```
string curl_unescape ( resource $ch , string $str )
```

curl_unescape() 函数解码给定的 URL 编码的字符串。

- $ch - 由 curl_init() 返回的 cURL 句柄。
- $str - 需要解码的 URL 编码字符串

```
<?php
// 创建一个 curl 句柄
$ch = curl_init('http://example.com/redirect.php');

// 发送 HTTP 请求并且遵循重定向
curl_setopt($ch, CURLOPT_FOLLOWLOCATION, 1);
curl_exec($ch);

// 获取最后的有效 URL
$effective_url = curl_getinfo($ch, CURLINFO_EFFECTIVE_URL);
// 结果: "http://example.com/show_location.php?loc=M%C3%BCnchen"

// 解码这个 URL
$effective_url_decoded = curl_unescape($ch, $effective_url);
// "http://example.com/show_location.php?loc=München"

// 关闭句柄
curl_close($ch);
```

注意，curl_unescape() 不会将加号 (+) 解码成空格，但是 urldecode() 会。

179.8 curl_exec()

执行一个 cURL 会话

```
mixed curl_exec ( resource $ch )
```

执行给定的 cURL 会话，而且这个函数应该在初始化一个 cURL 会话并且全部的选项都被设置后被调用。

- $ch - 由 curl_init() 返回的 cURL 句柄。

默认情况下，成功时返回 TRUE，或者在失败时返回 FALSE。不过，如果 CURLOPT_RETURNTRANSFER 选项被设置，函数执行成功时会返回执行的结果，失败时返回 FALSE 。

Example 181 curl_exec()

```
<?php
// 创建一个cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch, CURLOPT_HEADER, 0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close($ch);
```

179.9 curl_errno()

返回最后一次的错误号（如果没有错误发生，返回错误号为 0）。

```
int curl_errno ( resource $ch )
```

返回最后一次 cURL 操作的错误号。

- $ch - 由 curl_init() 返回的 cURL 句柄。

Example 182 curl_errno()

```
<?php
// 创建一个指向一个不存在的位置的cURL句柄
$ch = curl_init('http://404.php.net/');

// 执行
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
curl_exec($ch);

// 检查是否有错误发生
```

```
if(curl_errno($ch)) {
    echo 'Curl error: ' . curl_error($ch);
}

// 关闭句柄
curl_close($ch);
```

179.10 curl_error()

返回一个包含当前会话最近一次错误的字符串（如果没有任何错误发生，那么返回一个空字符串）

```
string curl_error ( resource $ch )
```

返回一条最近一次 cURL 操作明确的文本的错误信息。

- $ch - 由 curl_init() 返回的 cURL 句柄。

Example 183 curl_error()

```
<?php
// 创建一个指向一个不存在的位置的cURL句柄
$ch = curl_init('http://404.php.net/');
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

if(curl_exec($ch) === false){
    echo 'Curl error: ' . curl_error($ch);
} else {
    echo '操作完成没有任何错误';
}

// 关闭句柄
curl_close($ch);
```

179.11 curl_strerror()

Return string describing the given error code（如果错误码不存在则返回 NULL）

```
string curl_strerror ( int $errornum )
```

Returns a text error message describing the given error code.

- $errornum - cURL 错误码常量之一

179.12 curl_pause()

中止或继续一个连接（成功返回 CURLE_OK，出错则返回一个错误码）

```
int curl_pause ( resource $ch , int $bitmask )
```

- $ch - 由 curl_init() 返回的 cURL 句柄。
- CURLPAUSE_* 常量之一

```
<?php
// Create a curl handle with a mispelled protocol in URL
$ch = curl_init("htp://example.com/");

// Send request
curl_exec($ch);

// Check for errors and display the error message
if($errno = curl_errno($ch)) {
    $error_message = curl_strerror($errno);
    echo "cURL error ({$errno}):\n {$error_message}";
}

// Close the handle
curl_close($ch);
```

以上例程会输出：

```
cURL error (1):
 Unsupported protocol
```

179.13 curl_reset()

重置一个 libcurl 会话句柄的所有的选项

```
void curl_reset ( resource $ch )
```

curl_reset() 将给定的 cURL 句柄所有选项重新设置为默认值。

- $ch - 由 curl_init() 返回的 cURL 句柄。

```
<?php
// 创建一个url 句柄
$ch = curl_init();

// 设置 CURLOPT_USERAGENT 选项
```

```
curl_setopt($ch, CURLOPT_USERAGENT, "My test user-agent");

// 重置所有的预先设置的选项
curl_reset($ch);

// 发送 HTTP 请求
curl_setopt($ch, CURLOPT_URL, 'http://example.com/');
curl_exec($ch); // 预先设置的 user-agent 不会被发送，它已经被 curl_reset 重置掉了

// 关闭句柄
curl_close($ch);
```

注意，curl_reset() 同时也会重置 curl_init() 时传入的 URL 参数。

179.14 curl_close()

关闭一个 cURL 会话，没有返回值。

```
void curl_close ( resource $ch )
```

关闭一个 cURL 会话并且释放所有资源，而且 cURL 句柄 ch 也会被释放。

Example 184 初始化一个 cURL 会话来获取一个网页

```
<?php
// 创建一个新cURL资源
$ch = curl_init();

// 设置URL和相关选项
curl_setopt($ch,CURLOPT_URL,'http://example.com/');
curl_setopt($ch,CURLOPT_HEADER,0);

// 抓取URL并把它传递给浏览器
curl_exec($ch);

// 关闭cURL资源，并且释放系统资源
curl_close();
```

- $ch - 由 curl_init() 返回的 cURL 句柄。

179.15 curl_copy_handle()

复制一个 cURL 句柄和它的所有选项，并返回一个新的 cURL 句柄。

```
resource curl_copy_handle ( resource $ch )
```

复制一个 cURL 句柄并保持相同的选项。

- $ch - 由 curl_init() 返回的 cURL 句柄。

Example 185 复制一个 cURL 句柄

```
<?php
// 创建一个新的cURL资源
$ch = curl_init();

// 设置URL和相应的选项
curl_setopt($ch, CURLOPT_URL, 'http://www.example.com/');
curl_setopt($ch, CURLOPT_HEADER, 0);

// 复制句柄
$ch2 = curl_copy_handle($ch);

// 抓取URL (http://www.example.com/) 并把它传递给浏览器
curl_exec($ch2);

// 关闭cURL资源，并且释放系统资源
curl_close($ch2);
curl_close($ch);
```

179.16 curl_multi_init()

返回一个新 cURL 批处理句柄（失败则返回 FALSE）

```
resource curl_multi_init ( void )
```

cURL 允许并行地处理批处理 cURL 句柄。

下面的示例将会创建 2 个 cURL 句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建两个cURL资源
$ch1 = curl_init();
$ch2 = curl_init();
```

```
// 设置URL和相应的选项
curl_setopt($ch1,CURLOPT_URL,'http://www.example.com/');
curl_setopt($ch1,CURLOPT_HEADER,0);
curl_setopt($ch2,CURLOPT_URL,'http://www.php.net/');
curl_setopt($ch2,CURLOPT_HEADER,0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加两个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running = null;
// 执行批处理句柄
do {
   usleep(10000);
   curl_multi_exec($mh,$running);
}while(running > 0);

// 关闭全局句柄
curl_multi_remove_handle($mh,$ch1);
curl_multi_remove_handle($mh,$ch2);
curl_multi_close($mh);
```

179.17 curl_multi_info_read()

获取当前解析的 cURL 的相关传输信息（成功时返回相关信息的数组，失败时返回 FALSE）

```
array curl_multi_info_read ( resource $mh [, int &$msgs_in_queue = NULL ] )
```

查询批处理句柄是否单独的传输线程中有消息或信息返回。消息可能包含诸如从单独的传输线程返回的错误码或者只是传输线程有没有完成之类的报告。

重复调用这个函数，它每次都会返回一个新的结果，直到这时没有更多信息返回时，FALSE 被当作一个信号返回。

通过 msgs_in_queue 返回的整数指出将会包含当这次函数被调用后，还剩余的消息数。

返回的资源指向的数据在调用 curl_multi_remove_handle() 后将不会存在。

- $ch - 由 curl_multi_init() 返回的 cURL 多个句柄。
- $msgs_in_queue - 仍在队列中的消息数量。

返回数组的内容的具体描述如下：

表 179.10: curl_multi_info_read() 返回的数组的内容

key	value
msg	CURLMSG_DONE 常量。其他返回值当前不可用。
result	CURLE_* 常量之一。如果一切操作没有问题，将会返回 CURLE_OK 常量。
handle	cURL 资源类型表明它有关的句柄。

```
<?php
$urls = array(
   "http://www.cnn.com/",
   "http://www.bbc.co.uk/",
   "http://www.yahoo.com/"
);

$mh = curl_multi_init();

foreach ($urls as $i => $url) {
    $conn[$i] = curl_init($url);
    curl_setopt($conn[$i], CURLOPT_RETURNTRANSFER, 1);
    curl_multi_add_handle($mh, $conn[$i]);
}

do {
    $status = curl_multi_exec($mh, $active);
    $info = curl_multi_info_read($mh);
    if (false !== $info) {
        var_dump($info);
    }
} while ($status === CURLM_CALL_MULTI_PERFORM || $active);

foreach ($urls as $i => $url) {
    $res[$i] = curl_multi_getcontent($conn[$i]);
    curl_close($conn[$i]);
}

var_dump(curl_multi_info_read($mh));
```

以上例程的输出类似于：

```
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(5) of type (curl)
}
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(7) of type (curl)
}
array(3) {
  ["msg"]=>
  int(1)
  ["result"]=>
  int(0)
  ["handle"]=>
  resource(6) of type (curl)
}
bool(false)
```

179.18 curl_multi_add_handle()

向 curl 批处理会话中添加单独的 curl 句柄（增加 ch 句柄到批处理会话 mh）

```
int curl_multi_add_handle ( resource $mh , resource $ch )
```

- $mh - 由 curl_multi_init() 返回的 cURL 多个句柄。
- $ch - 由 curl_init() 返回的 cURL 句柄。

成功时返回 0，失败时返回 CURLM_XXX 之一的错误码。

下面的示例将会创建 2 个 cURL 句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
```

```
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running=null;
// 执行批处理句柄
do {
    curl_multi_exec($mh,$running);
} while($running > 0);

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);
```

179.19 curl_multi_remove_handle()

移除 curl 批处理句柄资源中的某个句柄资源（成功时返回一个 cURL 句柄，失败时返回 FALSE）

```
int curl_multi_remove_handle ( resource $mh , resource $ch )
```

从给定的批处理句柄 mh 中移除 ch 句柄。当 ch 句柄被移除以后，仍然可以合法地用 curl_exec() 执行这个句柄。当正在移除的句柄正在被使用，在处理的过程中所有的传输任务会被终止。

- $mh - 由 curl_multi_init() 返回的 cURL 多个句柄。
- $ch - 由 curl_init() 返回的 cURL 句柄。

179.20 curl_multi_setopt()

为 cURL 并行处理设置一个选项（成功时返回 TRUE，或者在失败时返回 FALSE）

```
bool curl_multi_setopt ( resource $mh , int $option , mixed $value )
```

- $mh
- $option - 常量 CURLMOPT_* 之一。
- $value - 将要设置给 option 的值。

在 option 参数为下列值时 value 需要为 int 类型：

表 179.11: cURL 的 curl_multi_setopt() 的 options 参数说明

Option 的值	将 value 设为
CURLMOPT_PIPELINING	传入 1 来启用或 0 来禁用。 在并行处理时启用管道模式将会尽可能地使用管线化的 HTTP （即 HTTP 长连接）来传输，这意味着如果你提交第二个请求，这个请求将会使用已经存在的链接，第二个请求将会被送入同一个链接的“管道”中。
CURLMOPT_MAXCONNECTS	传入一个数字来指定 libcurl 可以同时缓存的活跃链接的数量。默认值为 10。 当缓存写满时，lincurl 将关闭较早创建的链接来创建新的链接。

下面的示例说明如何同时启用 HTTP/1.1 pipelining 和 HTTP/2 multiplexing。

```
<?php
curl_multi_setopt($mh, CURLMOPT_PIPELINING, 3);
```

或

```
<?php
curl_multi_setopt($mh, CURLMOPT_PIPELINING, CURLPIPE_HTTP1 | CURLPIPE_MULTIPLEX);
```

179.21 curl_multi_select()

等待所有 cURL 批处理中的活动连接

```
int curl_multi_select ( resource $mh [, float $timeout = 1.0 ] )
```

阻塞直到 cURL 批处理连接中有活动连接。

- $mh - 由 curl_multi_init() 返回的 cURL 多个句柄。
- $timeout - 以秒为单位，等待响应的时间。

curl_multi_select() 成功时返回描述符集合中描述符的数量。失败时，select 失败时返回-1，否则返回超时 (从底层的 select 系统调用).

179.22 curl_multi_exec()

运行当前 cURL 句柄的子连接，并且返回一个定义于 cURL 预定义常量中的 cURL 代码。

```
int curl_multi_exec ( resource $mh , int &$still_running )
```

处理在栈中的每一个句柄。无论该句柄需要读取或写入数据都可调用此方法。

- $mh - 由 curl_multi_init() 返回的 cURL 多个句柄。
- $still_running - 一个用来判断操作是否仍在执行的标识的引用。

注意，curl_multi_exec() 函数仅返回关于整个批处理栈相关的错误。即使返回 CURLM_OK 时单个传输仍可能有问题。

下面的示例将会创建 2 个 cURL 句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://lxr.php.net/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$active = null;
// 执行批处理句柄
do {
    $mrc = curl_multi_exec($mh, $active);
} while ($mrc == CURLM_CALL_MULTI_PERFORM);

while ($active && $mrc == CURLM_OK) {
    if (curl_multi_select($mh) != -1) {
        do {
            $mrc = curl_multi_exec($mh, $active);
        } while ($mrc == CURLM_CALL_MULTI_PERFORM);
    }
}

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);
```

179.23 curl_multi_strerror()

返回错误码对应的错误信息（如果错误码已存在返回错误信息，否则返回 NULL）。

```
string curl_multi_strerror ( int $errornum )
```

Returns a text error message describing the given CURLM error code.

- $errornum - CURLM 错误代码常量之一

```
<?php
// Create cURL handles
$ch1 = curl_init("http://example.com"/);
$ch2 = curl_init("http://php.net/");

// Create a cURL multi handle
$mh = curl_multi_init();

// Add the handles to the multi handle
curl_multi_add_handle($mh, $ch1);
curl_multi_add_handle($mh, $ch2);

// Execute the multi handle
do {
    $status = curl_multi_exec($mh, $active);
    // Check for errors
    if($status > 0) {
        // Display error message
        echo "ERROR!\n " . curl_multi_strerror($status);
    }
} while ($status === CURLM_CALL_MULTI_PERFORM || $active);
```

179.24 curl_multi_getcontent()

如果设置了 CURLOPT_RETURNTRANSFER，则返回获取的输出的文本流

```
string curl_multi_getcontent ( resource $ch )
```

如果 CURLOPT_RETURNTRANSFER 作为一个选项被设置到一个具体的句柄，那么这个函数将会以字符串的形式返回那个 cURL 句柄获取的内容。

- $ch - 由 curl_init() 返回的 cURL 句柄。

179.25 curl_multi_close()

关闭一组 cURL 句柄

```
void curl_multi_close ( resource $mh )
```

- $mh - 由 curl_multi_init() 返回的 cURL 多个句柄。

下面的示例将会创建 2 个 cURL 句柄，把它们加到批处理句柄，然后并行地运行它们。

```
<?php
// 创建一对cURL资源
$ch1 = curl_init();
$ch2 = curl_init();

// 设置URL和相应的选项
curl_setopt($ch1, CURLOPT_URL, "http://www.example.com/");
curl_setopt($ch1, CURLOPT_HEADER, 0);
curl_setopt($ch2, CURLOPT_URL, "http://www.php.net/");
curl_setopt($ch2, CURLOPT_HEADER, 0);

// 创建批处理cURL句柄
$mh = curl_multi_init();

// 增加2个句柄
curl_multi_add_handle($mh,$ch1);
curl_multi_add_handle($mh,$ch2);

$running=null;
// 执行批处理句柄
do {
    curl_multi_exec($mh,$running);
} while ($running > 0);

// 关闭全部句柄
curl_multi_remove_handle($mh, $ch1);
curl_multi_remove_handle($mh, $ch2);
curl_multi_close($mh);
```

179.26 curl_share_init()

Initialize a cURL share handle（返回 cURL 共享句柄）

```
resource curl_share_init ( void )
```

cURL allows to share data between cURL handles.

下面的示例将创建一个 cURL 共享句柄，向其中添加两个 cURL 句柄，然后使用 cookie 数据共享运行它们。

```
<?php
// Create cURL share handle and set it to share cookie data
$sh = curl_share_init();
```

```
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// Initialize the first cURL handle and assign the share handle to it
$ch1 = curl_init("http://example.com/");
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// Execute the first cURL handle
curl_exec($ch1);

// Initialize the second cURL handle and assign the share handle to it
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// Execute the second cURL handle
//  all cookies from $ch1 handle are shared with $ch2 handle
curl_exec($ch2);

// Close the cURL share handle
curl_share_close($sh);

// Close the cURL handles
curl_close($ch1);
curl_close($ch2);
```

179.27 curl_share_setopt()

Set an option for a cURL share handle(成功时返回 TRUE，或者在失败时返回 FALSE).

```
bool curl_share_setopt ( resource $sh , int $option , string $value )
```

Sets an option on the given cURL share handle.

- $sh - A cURL share handle returned by curl_share_init().
- $option
    1. CURLSHOPT_SHARE - Specifies a type of data that should be shared.
    2. CURLSHOPT_UNSHARE - Specifies a type of data that will be no longer shared.
- $value
    1. CURL_LOCK_DATA_COOKIE - Shares cookie data.
    2. CURL_LOCK_DATA_DNS - Shares DNS cache. Note that when you use cURL multi handles, all handles added to the same multi handle will share DNS cache by default.
    3. CURL_LOCK_DATA_SSL_SESSION - Shares SSL session IDs, reducing the time spent on the SSL handshake when reconnecting to the same server. Note that SSL session IDs are reused within the same handle by default.

下面的示例将创建一个 cURL 共享句柄，向其中添加两个 cURL 句柄，然后使用 cookie 数据共享运行它们。

```
<?php
// Create cURL share handle and set it to share cookie data
```

```
$sh = curl_share_init();
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// Initialize the first cURL handle and assign the share handle to it
$ch1 = curl_init("http://example.com/");
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// Execute the first cURL handle
curl_exec($ch1);

// Initialize the second cURL handle and assign the share handle to it
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// Execute the second cURL handle
//  all cookies from $ch1 handle are shared with $ch2 handle
curl_exec($ch2);

// Close the cURL share handle
curl_share_close($sh);

// Close the cURL handles
curl_close($ch1);
curl_close($ch2);
```

179.28 curl_share_close()

Close a cURL share handle

```
void curl_share_close ( resource $sh )
```

Closes a cURL share handle and frees all resources.

- $sh - A cURL share handle returned by curl_share_init()

下面的示例将创建一个 cURL 共享句柄，向其中添加两个 cURL 句柄，然后使用 cookie 数据共享运行它们。

Example 186 关闭共享 cURL 句柄

```
<?php
// Create cURL share handle and set it to share cookie data
$sh = curl_share_init();
curl_share_setopt($sh, CURLSHOPT_SHARE, CURL_LOCK_DATA_COOKIE);

// Initialize the first cURL handle and assign the share handle to it
$ch1 = curl_init("http://example.com/");
```

```
curl_setopt($ch1, CURLOPT_SHARE, $sh);

// Execute the first cURL handle
curl_exec($ch1);

// Initialize the second cURL handle and assign the share handle to it
$ch2 = curl_init("http://php.net/");
curl_setopt($ch2, CURLOPT_SHARE, $sh);

// Execute the second cURL handle
//  all cookies from $ch1 handle are shared with $ch2 handle
curl_exec($ch2);

// Close the cURL share handle
curl_share_close($sh);

// Close the cURL handles
curl_close($ch1);
curl_close($ch2);
```

179.29 curl_file_create()

创建一个 CURLFile 对象，curl_file_create() 是此函数是 `CURLFile::__construct()` 的别名。

```
<?php
if (!function_exists('curl_file_create')) {
    function curl_file_create($filename, $mimetype = '', $postname = '') {
        return "@$filename;filename="
            . ($postname ?: basename($filename))
            . ($mimetype ? ";type=$mimetype" : '');
    }
}
```

Chapter 180

CURLFile Class

CURLFile 应该与选项 CURLOPT_POSTFIELDS 一同使用用于上传文件。

180.1 Synopsis

```
CURLFile {
   /* 属性 */
   public $name ;
   public $mime ;
   public $postname ;
   /* 方法 */
   public __construct ( string $filename [, string $mimetype [, string $postname ]] )
   public string getFilename ( void )
   public string getMimeType ( void )
   public string getPostFilename ( void )
   public void setMimeType ( string $mime )
   public void setPostFilename ( string $postname )
   public void __wakeup ( void )
}
```

180.2 Property

180.2.1 $name

$name - 待上传文件的名称

180.2.2 $mime

$mime - 文件的 MIME 类型（默认为 application/octet-stream）

180.2.3 $postname

$postname - 上传数据中的文件名称（默认为属性 name）

180.3 Method

180.3.1 CURLFile::__construct()

创建 CURLFile 对象（返回一个 CURLFile 对象）

- 面向对象风格

```
public CURLFile::__construct ( string $filename [, string $mimetype [, string
    $postname ]] )
```

- 过程式风格

```
CURLFile curl_file_create ( string $filename [, string $mimetype [, string
    $postname ]] )
```

创建一个 CURLFile 对象, 使用 CURLOPT_POSTFIELDS 选项上传文件。

- $filename - 被上传的文件的路径
- $mimetype - 被上传的文件的 MIME 类型
- $postname - 被上传文件的文件名

Example 187 CURLFile::__construct() 示例

```
<?php
/*
  * http://example.com/upload.php
  * var_dump($_FILES)
  */
// create a cURL handle
$ch = curl_init('http://example.com/upload.php');

// create a CURLFile object
$cfile = new CURLFile('test.jpg','image/jpeg','test_name');

// assign POST data
$data = array(
   'test_file'=>$cfile
);
curl_setopt($ch,CURLOPT_POST,1);
curl_setopt($ch,CURLOPT_POSTFIELDS,$data);

// execute the handle
curl_exec($ch);
```

Example 188 CURLFile::__construct() 示例 (过程化风格)

```
<?php
/*
  * http://example.com/upload.php
  * var_dump($_FILES)
  */
// Create a cURL handle
$ch = curl_init('http://example.com/upload.php');

// Create a CURLFile object
$cfile = curl_file_create('cats.jpg','image/jpeg','test_name');

// Assign POST data
$data = array('test_file' => $cfile);
curl_setopt($ch, CURLOPT_POST,1);
curl_setopt($ch, CURLOPT_POSTFIELDS, $data);

// Execute the handle
curl_exec($ch);
```

上述的示例的输出入如下:

```
array(1) {
  ["test_file"]=>
  array(5) {
    ["name"]=>
    string(9) "test_name"
    ["type"]=>
    string(10) "image/jpeg"
    ["tmp_name"]=>
    string(14) "/tmp/phpPC9Kbx"
    ["error"]=>
    int(0)
    ["size"]=>
    int(46334)
  }
}
```

180.3.2 CURLFile::getFilename()

获取被上传文件的文件名

```
public string CURLFile::getFilename ( void )
```

180.3.3 CURLFile::getMimetype()

获取被上传文件的 MIME 类型

```
public string CURLFile::getMimeType ( void )
```

180.3.4 CURLFile::getPostFilename()

获取 POST 请求时使用的文件名

```
public string CURLFile::getPostFilename ( void )
```

180.3.5 CURLFile::setMimeType()

设置被上传文件的 MIME 类型

```
public void CURLFile::setMimeType ( string $mime )
```

180.3.6 CURLFile::setPostFilename()

设置 POST 请求时使用的文件名

```
public void CURLFile::setPostFilename ( string $postname )
```

180.3.7 CURLFile::__wakeup()

反序列化句柄

```
public void CURLFile::__wakeup ( void )
```

注意，CURLFile 对象不能从序列化数据重建。

Chapter 181

CurlMulti

php-curlmulti 是目前最好的 php curl 类库，该类库是对 `curl_multi_*` 系列函数的封装。

- 极低的 CPU 和内存使用率。
- 速度在程序层面最高 (实测抓取 html 速度达到 2000+ 页每秒，下载速度 1000Mbps)。
- 支持全局并发设置和根据任务类型单独设置并发。
- 支持状态回调，运行中的所有信息都被返回，包括单独的每个任务信息。
- 支持通过回调添加任务。
- 支持用户自定义回调，可以在回调中做任何事情。
- 支持成功回调返回值控制任务。
- 支持全局错误回调和单独任务的错误回调，所有和错误相关的信息都被返回。
- 支持内部全自动重试。
- 支持用户参数任意传递。
- 支持 CURLOPT_* 全局设置和单个任务设置。
- 强大的内置缓存，可以设置全局缓存和单独任务缓存。
- 所有配置可以在运行中动态改变并生效!
- 基于 php-curlmulti 库可以开发各种强劲的 CURL 应用。

PHP 周边性能优化的推荐规则如下：

- Linux 运行环境，服务器环境切换到 Linux；
- 文件存储 (硬盘)
- 数据库
- 缓存 (内存)
- 网络
- 减少文件类操作
- 减少 PHP 网络操作 (设置超时时间：1. 连接超时 200ms 2. 读超时 800ms 3. 写超时 500ms)
- 将串行请求并行化，使用 curl_multi_*() 或使用 swoole 扩展实现
- 压缩 PHP 接口输出 (100K 以上使用 gzip，利：输出的字符串少了，更快的接收，弊：额外的 cpu 开销)
- PHP 缓存复用
- Smarty 调优和重叠时间窗口思想
- 借助 XHProf 工具分析 PHP 性能 (PHP –ri 扩展名)
- PHP 性能分析工具扩展 1.ab 压力测试 2.vld opcode 代码分析
- PHP 性能瓶颈 1.opcode cache：PHP 扩展 APC；2.runtime 优化：HHVM
- pecl.php.net 提供 PHP 的扩展

181.1 Composer

基于 composer 安装 php-curlmulti 的命令如下:

```
$ composer require phpdr.net/php-curlmulti:2.*
```

181.2 Pthreads

没有 pthreads 扩展支持的情况下，PHP 是单线程顺序执行的，所以 php-curlmulti 类库大量使用回调函数。

181.3 Method

php-curlmulti 类库本身只有两个常用的方法，add() 和 start()，其中:

- add() 添加一个任务到内部任务池;
- start() 开始以 $maxTrhead 设置的并发数进行回调循环，此方法是阻塞的直到所有任务完成。

如果有大量的任务需要处理，使用 $cbTask 指定添加任务的回调函数，当并发不足并且任务池为空时此回调函数被调用。

181.3.1 add()

当一个任务完成之后 add() 中指定的回调立刻被执行，然后 curl 从任务池取一个任务添加到并发请求中。

181.3.2 start()

所有任务完成后 start() 函数结束。

181.4 Source

181.4.1 src/Core.php

src/Core.php 是 php-curlmulti 的核心库。

181.4.2 src/Base.php

src/Base.php 是 php-curlmulti 核心库的封装，包含非常有用的工具和一些规范。

181.4.3 src/Exception.php

src/Exception.php 是 php-curlmulti 的异常处理类。

181.4.4 src/Autoclone.php

src/Autoclone.php 是一个完美的全自动网站克隆工具。

- 软件工程，面向对象和编程技巧的完美结晶，和 php-curlmulti 一样非常具有艺术性。
- 使用方便，只有一个无参方法 start()。
- 低耦合，扩展极其容易，配合 php-curlmulti 的强大能力可以分分钟拷贝一个站。
- 所有页面的重复的 url 只会精确处理一次。

- 全自动处理任意格式 url 的相对路径绝对路径。
- css 中引入的背景图等资源全自动处理，css 中的 @import 全自动处理，支持任意深度!
- 支持指定多个前缀 url 并且可以针对每个前缀 url 设置一个深度。
- 支持对每个前缀 url 指定二级前缀，每个二级前缀还可以设置一个深度。
- 全自动处理 3xx 跳转。
- 跨站资源共享，例如采集站点 A 的时候 A 用到了站点 B 的图片，jss，css 等，等采集站点 B 的时候这些文件会直接使用不会再次下载。
- 同一个目录下可以复制任意数量的站点并且不会发生任何可能的文件重复或覆盖。
- 支持不同类型资源文件下载控制。

针对 IE 浏览器的注释性 css 不会处理，因为还没找到一种符合标准的处理方式，这个问题造成的影响可以忽略不计。

181.5 API(Core)

```
public $maxThread = 10
```

```
public $maxThreadType = array()
```

```
public $maxTry = 3
```

```
public $opt = array ()
```

```
public $cache = array ('enable' => false, 'enableDownload'=> false, 'compress' => false,
     'dir' => null, 'expire' =>86400, 'dirLevel' => 1, 'verifyPost' => false, 'overwrite
    ' => false, 'overwriteExpire' => 86400)
```

```
public $taskPoolType = 'queue'
```

```
public $cbTask = array(0=>'callback',1=>'callback param')
```

```
public $cbInfo = null
```

```
public $cbUser = null
```

```
public $cbFail = null
```

```
public function __construct()
```

```
public function add(array $item, $process = null, $fail = null)
```

- `$item['url']` 不能为空。
- `$item['opt']=array()` 当前任务的 CURLOPT_*，覆盖全局的 CURLOPT_*。
- `$item['args']` 成功和失败回调的第二个参数。
- `$item['ctl']=array()` 一些额外的控制项
- `$item['ctl']['type']` 任务类型，用在 $maxThreadType 属性。
- `$item['ctl']['cache']=array()` 任务缓存配置，覆盖合并 $cache 属性。
- `$item['ctl']['ahead']` 忽略 $taskPoolType 属性，此种类型的任务总是被优先加入并发中。
- `$process` 任务成功完成调用此回调，回调的第一个参数是结果数组，第二个参数是 `$item['args']`。
- `$fail` 任务失败回调，第一个参数是相关信息，第二个参数是 `$item['args']`。

```
public function start($persist=null)
```

181.6 API(Base)

```
function __construct($curlmulti = null)

function hashpath($name, $level = 2)

function substr($str, $start, $end = null, $mode = 'g')

function cbCurlFail($error, $args)

function cbCurlInfo($info,$isFirst,$isLast)

function encoding($html, $in = null, $out = 'UTF-8', $mode = 'auto')

function isUrl($str)

function uri2url($uri, $urlCurrent)

function url2uri($url, $urlCurrent)

function urlDir($url)

function getCurl()
```

Part XLVI

Date

Chapter 182

Overview

Part XLVII

DOM

Chapter 183

Overview

Part XLVIII

Enchant

Chapter 184

Overview

Part XLIX

Exif

Chapter 185

Overview

Part L

Fileinfo

Chapter 186

Overview

Part LI

Filter

Chapter 187

Overview

Part LII

FTP

Chapter 188

Overview

Part LIII

GD

Chapter 189

Overview

PHP 并不仅限于创建 HTML 输出，它也可以创建和处理包括 GIF，PNG，JPEG，WBMP 以及 XPM 在内的多种格式的图像。

更加方便的是，PHP 可以直接将图像数据流输出到浏览器。要想在 PHP 中使用图像处理功能，你需要连带 GD 库一起来编译 PHP。GD 库和 PHP 可能需要其他的库，这取决于你要处理的图像格式。

你可以使用 PHP 中的图像函数来获取下列格式图像的大小：JPEG，GIF，PNG，SWF，TIFF 和 JPEG2000。

如果联合 exif 扩展一起使用，你可以操作存储在 JPEG 和 TIFF 图像文件头部的信息，这样就就可以获取数码相机所产生的元数据。exif 相关的函数不需要 GD 库亦可使用。

Part LIV

Gearman

Chapter 190

Overview

Part LV

GeoIP

Chapter 191

Overview

Part LVI

GetText

Chapter 192

Overview

Part LVII

GMagick

Chapter 193

Overview

Part LVIII

GMP

Chapter 194

Overview

Part LIX

GnuPG

Chapter 195

Over

Part LX

Hash

Chapter 196

Overview

Part LXI

HTTP

Chapter 197

Overview

```
$ php --ri http
http

HTTP Support => enabled
Extension Version => 3.1.0

Used Library => Compiled => Linked
libz => 1.2.8 => 1.2.8
libcurl => 7.35.0 => 7.35.0
libevent => 2.0.21-stable => 2.0.21-stable
libicu (IDNA2008/IDNA2003) => 52.1 => 52.1
libidn2 (IDNA2008) => disabled => disabled
libidn (IDNA2003) => 1.28 => unknown
libidnkit2 (IDNA2008) => disabled => disabled
libidnkit (IDNA2003) => disabled => disabled

Directive => Local Value => Master Value
http.etag.mode => crc32b => crc32b
```

Part LXII

ICONV

Chapter 198

Overview

Part LXIII

Igbinary

Chapter 199

Overview

Part LXIV

Imagick

Chapter 200

Overview

Part LXV

Inotify

Chapter 201

Overview

inotify 扩展提供了若干提供通知的函数，例如 inotify_init()、inotify_add_watch() 和 inotify_rm_watch() 等。

和 C 语言的 inotify_init() 返回一个文件描述符不同，PHP 的 inotify_init() 函数返回一个流资源，可以被标准流函数（例如 stream_select()、stream_set_blocking() 和 fclose() 等）使用。

inotify_read() 函数替换了 C 语言读取通知事件的方式。

201.1 Requirement

inotify 扩展需要较新的 Linux 内核和 C 标准库。

201.2 Resource Type

inotify 扩展的 inotify_init() 函数返回一个流资源。

201.3 Runtime Configure

inotify 扩展没有在 php.ini 中定义配置指令。

201.4 Predefined Constants

下列常量由 inotify 扩展定义，且仅在该扩展编译入 PHP 或在运行时动态载入时可用。

Inotify constants usable with inotify_add_watch() and/or returned by inotify_read()

上面标有星号（*）的事件可能发生在监视目录中的文件。

201.5 Functions

201.5.1 inotify_add_watch()

201.5.2 inotify_init()

201.5.3 inotify_queue_len()

201.5.4 inotify_read()

201.5.5 inotify_rm_watch()

Part LXVI

Mailparse

Chapter 202

Overview

Part LXVII

Mbstring

Chapter 203

Overview

Part LXVIII

Mcrypt

Chapter 204

Overview

Part LXIX

Memcache

Chapter 205

Overview

Part LXX

Memcached

Chapter 206

Overview

Part LXXI

MongoDB

Chapter 207

Overview

Part LXXII

Msgpack

Chapter 208

Overview

```
$ sudo pecl install msgpack
```

```
$ /path/to/phpize
$ ./configure
$ make && make install
```

```
<?php
$data = array(0=>1,1=>2,2=>3);
$msg = msgpack_pack($data);
$data = msgpack_unpack($msg);
```

Part LXXIII

Oauth

Chapter 209

Overview

Part LXXIV

PCRE

Chapter 210

Overview

Part LXXV

PCS

Chapter 211

Overview

Part LXXVI

PDO

Chapter 212

Overview

PDO（PHP Data Objects）是一个可以用作数据库抽象层的 PHP 扩展。

Part LXXVII

PDO_MySQL

Chapter 213

Overview

Part LXXVIII

PDO_PGSQL

Chapter 214

Overview

Part LXXIX

Phar

Chapter 215

Overview

Part LXXX

Propro

Chapter 216

Overview

Part LXXXI

Radius

Chapter 217

Overview

Part LXXXII

Raphf

Chapter 218

Overview

Part LXXXIII

Readline

Chapter 219

Overview

Part LXXXIV

Redis

Chapter 220

Overview

Part LXXXV

Reflection

Chapter 221

Overview

Part LXXXVI

RRD

Chapter 222

Overview

Part LXXXVII

SeasLog

Chapter 223

Overview

223.1 Session

Chapter 224

Overview

Part LXXXVIII

Sockets

Chapter 225

Overview

Part LXXXIX

Sysvmsg

Chapter 226

Overview

Part XC

Sysvsem

Chapter 227

Overview

Part XCI

Sysvshm

Chapter 228

Overview

Part XCII

Tidy

Chapter 229

Overview

Part XCIII

Tokenizer

Chapter 230

Overview

Part XCIV

UUID

Chapter 231

Overview

Part XCV

WDDX

Chapter 232

Overview

Part XCVI

XMLReader

Chapter 233

Overview

Part XCVII

XMLWriter

Chapter 234

Overview

Part XCVIII

XSL

Chapter 235

Overview

Part XCIX

Yaml

Chapter 236

Overview

Part C

Zip

Chapter 237

Overview

Part CI

Zlib

Chapter 238

Overview

Part CII

ZMQ

Chapter 239

Overview

ZeroMQ 是一个允许用户快速设计和实现快速的基于消息的应用程序的软件库。

- 支持任何语言和任何平台连接消息系统
- 支持在进程内、IPC、TCP、TIPC 和广播来传递信息
- 支持 publish/subscribe、push/pull 和 router/dealer
- 内置的库可以提供高速的异步 I/O 引擎
- 支持构建中心式、分布式的不同规模的架构

```
<?php

/* Create new queue object */
$queue = new ZMQSocket(new ZMQContext(), ZMQ::SOCKET_REQ, "MySock1");

/* Connect to an endpoint */
$queue->connect("tcp://127.0.0.1:5555");

/* send and receive */
var_dump($queue->send("hello there, using socket 1")->recv());
```

239.1 Requirement

PECL/zmq 需要有 ZeroMQ 支持。

```
$ ./configure
$ make
$ make install
Making install in doc
make[1]: Entering directory `/home/zgy/Downloads/zeromq-4.2.0/doc'
make[2]: Entering directory `/home/zgy/Downloads/zeromq-4.2.0/doc'
make[2]: Nothing to be done for `install-exec-am'.
 /bin/mkdir -p '/usr/local/share/man/man3'
```

```
 /usr/bin/install -c -m 644 zmq_bind.3 zmq_unbind.3 zmq_connect.3 zmq_disconnect.3
     zmq_close.3 zmq_ctx_new.3 zmq_ctx_term.3 zmq_ctx_get.3 zmq_ctx_set.3
     zmq_ctx_shutdown.3 zmq_msg_init.3 zmq_msg_init_data.3 zmq_msg_init_size.3
     zmq_msg_move.3 zmq_msg_copy.3 zmq_msg_size.3 zmq_msg_data.3 zmq_msg_close.3
     zmq_msg_send.3 zmq_msg_recv.3 zmq_msg_routing_id.3 zmq_msg_set_routing_id.3
     zmq_send.3 zmq_recv.3 zmq_send_const.3 zmq_msg_get.3 zmq_msg_set.3 zmq_msg_more.3
     zmq_msg_gets.3 zmq_getsockopt.3 zmq_setsockopt.3 zmq_socket.3 zmq_socket_monitor.3
     zmq_poll.3 zmq_errno.3 zmq_strerror.3 zmq_version.3 zmq_sendmsg.3 zmq_recvmsg.3
     zmq_proxy.3 '/usr/local/share/man/man3'
 /usr/bin/install -c -m 644 zmq_proxy_steerable.3 zmq_z85_encode.3 zmq_z85_decode.3
     zmq_curve_keypair.3 zmq_has.3 zmq_atomic_counter_new.3 zmq_atomic_counter_set.3
     zmq_atomic_counter_inc.3 zmq_atomic_counter_dec.3 zmq_atomic_counter_value.3
     zmq_atomic_counter_destroy.3 '/usr/local/share/man/man3'
 /bin/mkdir -p '/usr/local/share/man/man7'
 /usr/bin/install -c -m 644 zmq.7 zmq_tcp.7 zmq_pgm.7 zmq_inproc.7 zmq_ipc.7 zmq_null.7
     zmq_plain.7 zmq_curve.7 zmq_tipc.7 zmq_vmci.7 zmq_udp.7 '/usr/local/share/man/man7'
make[2]: Leaving directory `/home/zgy/Downloads/zeromq-4.2.0/doc'
make[1]: Leaving directory `/home/zgy/Downloads/zeromq-4.2.0/doc'
make[1]: Entering directory `/home/zgy/Downloads/zeromq-4.2.0'
make[2]: Entering directory `/home/zgy/Downloads/zeromq-4.2.0'
 /bin/mkdir -p '/usr/local/lib'
 /bin/bash ./libtool --mode=install /usr/bin/install -c src/libzmq.la '/usr/local/lib'
libtool: install: /usr/bin/install -c src/.libs/libzmq.so.5.1.0 /usr/local/lib/libzmq.so
    .5.1.0
libtool: install: (cd /usr/local/lib && { ln -s -f libzmq.so.5.1.0 libzmq.so.5 || { rm -
    f libzmq.so.5 && ln -s libzmq.so.5.1.0 libzmq.so.5; }; })
libtool: install: (cd /usr/local/lib && { ln -s -f libzmq.so.5.1.0 libzmq.so || { rm -f
    libzmq.so && ln -s libzmq.so.5.1.0 libzmq.so; }; })
libtool: install: /usr/bin/install -c src/.libs/libzmq.lai /usr/local/lib/libzmq.la
libtool: install: /usr/bin/install -c src/.libs/libzmq.a /usr/local/lib/libzmq.a
libtool: install: chmod 644 /usr/local/lib/libzmq.a
libtool: install: ranlib /usr/local/lib/libzmq.a
libtool: finish: PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr
    /games:/usr/local/games:/usr/local/texlive/bin/x86_64-linux:/sbin" ldconfig -n /usr/
    local/lib
Libraries have been installed in:
   /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
```

```
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,-rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
 /bin/mkdir -p '/usr/local/include'
 /usr/bin/install -c -m 644 include/zmq.h include/zmq_utils.h '/usr/local/include'
 /bin/mkdir -p '/usr/local/lib/pkgconfig'
 /usr/bin/install -c -m 644 src/libzmq.pc '/usr/local/lib/pkgconfig'
make[2]: Leaving directory `/home/zgy/Downloads/zeromq-4.2.0'
make[1]: Leaving directory `/home/zgy/Downloads/zeromq-4.2.0'
```

Chapter 240

ZMQ Class

240.1 Class Synopsis

```
ZMQ {
/* Constants */
const integer SOCKET_PAIR ;
const integer SOCKET_PUB ;
const integer SOCKET_SUB ;
const integer SOCKET_REQ ;
const integer SOCKET_REP ;
const integer SOCKET_XREQ ;
const integer SOCKET_XREP ;
const integer SOCKET_PUSH ;
const integer SOCKET_PULL ;
const integer SOCKET_ROUTER ;
const integer SOCKET_DEALER ;
const integer SOCKET_XPUB ;
const integer SOCKET_XSUB ;
const integer SOCKET_STREAM ;
const integer SOCKOPT_HWM ;
const integer SOCKOPT_SNDHWM ;
const integer SOCKOPT_RCVHWM ;
const integer SOCKOPT_AFFINITY ;
const integer SOCKOPT_IDENTITY ;
const integer SOCKOPT_SUBSCRIBE ;
const integer SOCKOPT_UNSUBSCRIBE ;
const integer SOCKOPT_RATE ;
const integer SOCKOPT_RECOVERY_IVL ;
```

```
const integer SOCKOPT_RECONNECT_IVL ;
const integer SOCKOPT_RECONNECT_IVL_MAX ;
const integer SOCKOPT_MCAST_LOOP ;
const integer SOCKOPT_SNDBUF ;
const integer SOCKOPT_RCVBUF ;
const integer SOCKOPT_RCVMORE ;
const integer SOCKOPT_TYPE ;
const integer SOCKOPT_LINGER ;
const integer SOCKOPT_BACKLOG ;
const integer SOCKOPT_MAXMSGSIZE ;
const integer SOCKOPT_SNDTIMEO ;
const integer SOCKOPT_RCVTIMEO ;
const integer SOCKOPT_IPV4ONLY ;
const integer SOCKOPT_LAST_ENDPOINT ;
const integer SOCKOPT_TCP_KEEPALIVE_IDLE ;
const integer SOCKOPT_TCP_KEEPALIVE_CNT ;
const integer SOCKOPT_TCP_KEEPALIVE_INTVL ;
const integer SOCKOPT_TCP_ACCEPT_FILTER ;
const integer SOCKOPT_TCP_ACCEPT_FILTER ;
const integer SOCKOPT_DELAY_ATTACH_ON_CONNECT ;
const integer SOCKOPT_XPUB_VERBOSE ;
const integer SOCKOPT_ROUTER_RAW ;
const integer SOCKOPT_IPV6 ;
const integer CTXOPT_MAX_SOCKETS ;
const integer POLL_IN ;
const integer POLL_OUT ;
const integer MODE_NOBLOCK ;
const integer MODE_DONTWAIT ;
const integer MODE_SNDMORE ;
const integer ERR_INTERNAL ;
const integer ERR_EAGAIN ;
const integer ERR_ENOTSUP ;
const integer ERR_EFSM ;
const integer ERR_ETERM ;
/* Methods */
private__construct ( void )
}
```

240.2 Predefined Constants

Chapter 241

ZMQContext Class

241.1 Class Synopsis

```
ZMQContext {
/* Methods */
__construct ([ integer $io_threads = 1 [, boolean $is_persistent = true ]] )
public mixed getOpt ( string $key )
public ZMQSocket getSocket ( integer $type [, string $persistent_id = null [, callback
    $on_new_socket = null ]] )
public boolean isPersistent ( void )
public ZMQContext setOpt ( integer $key , mixed $value )
}
```

Chapter 242

ZMQSocket Class

242.1 Class Synopsis

```
ZMQSocket {
/* Methods */
public ZMQSocket bind ( string $dsn [, boolean $force = false ] )
public ZMQSocket connect ( string $dsn [, boolean $force = false ] )
__construct ( ZMQContext $context , int $type [, string $persistent_id = null [,
    callback $on_new_socket = null ]] )
public ZMQSocket disconnect ( string $dsn )
public array getEndpoints ( void )
public string getPersistentId ( void )
public integer getSocketType ( void )
public mixed getSockOpt ( string $key )
public boolean isPersistent ( void )
public string recv ([ integer $mode = 0 ] )
public string recvMulti ([ integer $mode = 0 ] )
public ZMQSocket send ( string $message [, integer $mode = 0 ] )
public ZMQSocket send ( array $message [, integer $mode = 0 ] )
public ZMQSocket setSockOpt ( integer $key , mixed $value )
public ZMQSocket unbind ( string $dsn )
}
```

Chapter 243

ZMQPoll Class

243.1 Class Synopsis

```
ZMQPoll {
/* Methods */
public string add ( mixed $entry , integer $type )
public ZMQPoll clear ( void )
public integer count ( void )
public array getLastErrors ( void )
public integer poll ( array &$readable , array &$writable [, integer $timeout = -1 ] )
public boolean remove ( mixed $item )
}
```

Chapter 244

ZMQDevice Class

244.1 Class Synopsis

```
ZMQDevice {
/* Methods */
__construct ( ZMQSocket $frontend , ZMQSocket $backend [, ZMQSocket $listener ] )
public ZMQDevice getIdleTimeout ( void )
public ZMQDevice getTimerTimeout ( void )
public void run ( void )
public ZMQDevice setIdleCallback ( callable $cb_func , integer $timeout [, mixed
    $user_data ] )
public ZMQDevice setIdleTimeout ( integer $timeout )
public ZMQDevice setTimerCallback ( callable $cb_func , integer $timeout [, mixed
    $user_data ] )
public ZMQDevice setTimerTimeout ( integer $timeout )
}
```

Part CIII

Zend Opcache

Chapter 245

Overview

Part CIV

PHPUnit

Chapter 246

Overview

为 PHP 程序编写自动化测试被认为是最佳实践，可以建立良好的应用程序。其中，自动化测试是非常棒的工具，它能确保你的应用程序在改变或增加新的功能时不会影响现有的功能，不应该忽视。

PHP 有一些不同种类的测试工具 (或框架) 可以使用，它们使用不同的方法，但是它们都试图避免手动测试和大型 QA 团队的需求，确保最近的变更不会破坏既有功能。

测试驱动开发 (TDD) 是一种以非常短的开发周期不断迭代的软件开发过程，首先开发者对将要实现的功能或者新的方法写一个失败的自动化测试用例，然后就去写代码来通过这个测试用例，最终通过重构代码让一其达到可接受的水准。

在 PHP 应用开发中可以应用的测试类型包括单元测试、集成测试、功能性测试和其他测试。

除了个别的测试驱动和行为驱动框架之外，还有一些通用的框架和辅助函数类库，对任何的测试方法都很有用。例如，Selenium 是一个集成了 PHPUnit 的浏览器自动化测试工具，Mockery 是一个可以跟 PHPUnit 或者 PHPSpec 整合的 Mock 对象框架，Prophecy 是个有自己的想法并且非常强大灵活的 PHP 对象 mocking 框架，它整合了 PHPSpec 并且可以和 PHPUnit 一起使用。

246.1 Requirement

PHPUnit 需要 dom、json、pcre、phar、openssl、Reflection、Spl、Xdebug、tokenizer 和 xmlwriter 的支持。

246.2 Unit Test

单元测试是一种编程方法来确认函数、类和方法以开发者预期的方式来工作，而且单元测试会贯穿整个项目的开发周期。

开发者通过单元测试检查各个函数和方法的输入输出，就可以保证内部的逻辑已经正确执行。通过使用依赖注入和编写”mock”类以及 stubs 来确认依赖被正确的使用，提高测试覆盖率。

当创建一个类或者一个函数时，应该为它们的每一个行为创建一个单元测试。至少应该确认当用户输入一个错误参数会触发一个错误，而且输入一个有效的参数会得到正确的结果，这样就会帮助开发者在开发周期后段对类或者函数做出修改后，确认已有的功能仍然可以正常的工作。

可替代的方法是在源码中使用 var_dump() ，但是这种方法不能用户构建一个或大或小的应用。

单元测试的其他用处是在给开源项目贡献代码时。如果提交者写了一个测试证明代码有 bug，然后修复它，并且展示测试的过程，这样补丁将会更容易被接受。如果在维护一个项目，在处理 pull request 的时候可以将单元测试作为一个要求。

除了 PHP 应用开发单元测试框架的标准 PHPUnit 之外，也有其他可选的框架，例如 atoun、Enhance PHP、PUnit 和 SimpleTest 等。

246.2.1 PHPUnit

246.3 Integration Test

集成测试 (集成和测试，缩写为 I&T) 是把各个模块组合在一起进行整体测试的软件测试阶段。

集成测试处于单元测试之后，验收测试之前。集成测试将已经经过了单元测试的模块做为输入模块，组合成一个整体，然后运行集成测试用例，最终输出一个可以进行系统测试的系统。

许多相同的测试工具既可以运用到单元测试，也可以运用到集成测试。

246.4 Acceptance Test

验收测试（或功能测试）是通过使用工具来生成自动化的测试用例，然后在真实的系统上运行，而不是单元测试中简单的验证单个模块的正确性和集成测试中验证各个模块间交互的正确性。

验收测试的相关工具（例如 Selenium、Mink、Codeception 和 Storyplayer 等）会使用代表性的真实数据来模拟真实用户的行为来验证系统的正确性。

- Codeception 是一个全栈的测试框架包括验收性测试工具。
- Storyplayer 是一个全栈的测试框架并且支持随时创建和销毁测试环境。

246.4.1 Codeception

246.4.2 Storyplayer

246.5 Behavior Driven

有两种不同的行为驱动开发 (BDD)——SpecBDD 和 StoryBDD，其中 SpecBDD 专注于代码的技术行为，而 StoryBDD 专注于业务逻辑或功能的行为和互动，这两种 BDD 都有对应的 PHP 框架。

采用 StoryBDD 时，开发者可以编写可读的故事来描述应用程序的行为。接着这些故事可以作为应用程序的实际测试案例执行。Behat 就是使用在 PHP 应用程序中的 StoryBDD 框架，它受到 Ruby 的 Cucumber 项目的启发并且实现了 Gherkin DSL 來描述功能的行为。

采用 SpecBDD 时，开发者编写规格来描述实际的代码应该有什么行为。你应该描述函数或者方法应该有什么行为，而不是测试函数或者方法。PHP 提供了 PHPSpec 框架来达到这个目的，这个框架受到了 Ruby 的 RSpec project 项目的启发。

- Behat 是 PHP 的 StoryBDD 框架，受到了 Ruby’s Cucumber 项目的启发。
- PHPSpec 是 PHP 的 StoryBDD 框架，受到了 Ruby’s Cucumber 项目的启发。
- PHP 的 StoryBDD 框架，受到了 Ruby’s Cucumber 项目的启发。

246.5.1 Behat

246.5.2 PHPSpec

246.5.3 StoryBDD

Part CV

Code Sniffer

Chapter 247

Overview

Part CVI

Domain-Driven Design

Chapter 248

Overview

“领域驱动设计”，顾名思义，首先强调的是“领域”。这个词不是指技术上的任何东西，而是指“业务领域”，是说用领域的角度去分析和设计业务。

现实中的大部分开发者并不会直接接触业务，实际接触到的仅仅是架构师消化过的，而且企业开发中处处充满着管理的不规范性，再加上用户的很多需求本身就是不规范的，因此实际生产出的软件只能是怪胎，大多数软件只能满足一时之需，跟不上用户的成长和规范。

为了实施领域驱动设计，需要在向领域专家和专业书籍学习的同时，找出当前用户在业务不足之处，从而在规范和现实之间构建桥梁。

248.1 MDD

MDD（Model-Driven Development）包括基于图形（MDE）和基于文本语言（DSL）等形式，MDD 的建议：

- 尽量保持领域模型更加紧凑和窄化。
- 领域要众所周知。
- 对 MDD 要足够重视，不要作为试点项目。
- MDD 最好形式是落地
- Be careful of gains that are offset elsewhere
- 不要对代码生成迷惑
- 不是每个人能够抽象思考。
- 大多数项目失败在可伸缩上。
- 使用正确的工具和过程。

图形化在架构基本虚拟抽象是有用处的，但是在对象组件构件级别没有任何意义，相比基于文本语言，图形化工具难于和环境以及开发过程进行整合。

文本建模语言 DSL 已经有不少成功工业应用，当然还有些问题（比如削弱 MDD 的一些特点），好的编程语言不但在细节上执行有效，而且支持 DSL 构建，提供抽象层次，并且可以用在组件和类似 MDD 的任务上。

所有 MDD 都是基于面向对象，但是，对象在构建可重用的组件构件标准上力不从心，已知的成功的标准组件（如 TCP 和 HTTP 等）都不是对象化的，而面向对象的组件标准（如 CORBA）则证明会带来复杂性。

函数式编程语言则具备了构成成功组件的构件标准的特点。

248.2 Entity

过去看到的 OO 都是“简单实体建模”，也就是从需求中挖出几个实体对象，然后填充属性，至于其它的只要会查询即可，这种建模方式随处可见。

其实 OO 就是“贫血对象”，DDD 之所以让领域对象富有生命，是因为值对象的存在。DDD 采取的是“特征建模”思想，领域对象不仅仅有实体对象，还有表示状态的值对象。

DDD 强调“充血模型”。在复杂业务中，这很有用，可以提高对象的可复用性。但是要注意，“充血模型”虽然很好，但其背后却隐藏玄机。复杂场景中，如果所有的行为都放到了对象中，那领域对象就会变得沉重无比，带来各种副作用。同时，描述状态的值对象往往不是一蹴而就，而是充满着变化，难以把握。当业务不明时，”贫血模型”容易把握，这也就是为什么现在”充血模型“不多，而”贫血模型”大行其道的原因。

DDD 认为”领域对象应该是有行为的”，大家都认可。如果说”贫血模型”造成了对象和行为的分离的话；那么”充血模型”也是有问题的，因为 DDD 中简单地认为把行为放到对象中就行了，殊不知这其实是关于对象和行为之间的一种静态地僵化的看法。

DCI 对这种思想进行了修正，认为对象在场景中才具有行为。DCI 中的对象（数据）是一种裸对象，但却不是简单意义上的“贫血对象”。

DCI 中的对象和行为是一种动态关系，是依赖于场景而存在的，这也就是说对象不会无缘无故的产生行为，而且对象具有行为一定是有意义的。

在 DCI 中可以用”事件”把对象和行为联系在一起就避免了”贫血对象”的问题。

另外，DDD 提出聚合的概念是为了保证领域内对象之间的一致性问题，DDD 特别强调聚合根的封装性，不过可能会导致领域内对象之间的逻辑强耦合。

在实战中，领域模型内部的值对象往往存在着变数，这是我们认识客观世界的必然规律。然而这会导致领域模型的不稳定性。所以即使领域内部的对象也应该注意低耦合，这个问题同样需要靠事件来解决，事件才是保证领域对象一致性的关键。

领域聚合内部的松耦合因为天然的业务的高聚合性只能通过软件设计上的松耦合方式来实现（比如设计模式等）。

为了保证一致性，一般采取在实体根对象中放入对值对象或其他状态实体修改的方法行为，外界不可以绕过根实体直接修改状态。

另外，值对象并不适合保存状态，状态应该属于实体的一个重要部分，如果状态复杂，可以用专门的状态模式等设计方式消肿，所以担心“充血模型”会变得沉重无比是多余的，因为这里面忽视了软件设计模式的作用。

总体来说，根据业务的设计和根据纯软件设计目标松耦合的设计这两种切分方式，可以将业务真正落实为好的软件，而我们过去常常注重业务的切分设计，忽视软件层面设计，也就是设计模式。

当然，掌握设计模式的人去进行业务分析设计时，发现设计模式毫无用处，这是因为没有掌握业务建模设计的原因，业务建模设计好了，自然软件设计就派上用上。

业务建模设计和软件设计相当于切菜的大刀和小刀，对于整只猪，必须先用大刀，然后再用小刀。有了这两把刀，就不用担心胖对象的沉重，相反，如果一开始做成很瘦的只有数据的贫血模型，设计模式真的一无用处了，然后系统开发就走上了依赖数据库/数据表的系统，根本没有设计。

248.3 Problem

现实业务中的问题域模型更侧重强调是“领域模型”，因此很难立体化并动态地完整描述这个“问题域模型”，技术的演进目的是试图解决“问题域模型”的变化问题。

“领域模型”存在与”问题域模型“不匹配的问题，原因可能是时间变化或者用户场景变化导致的，因此对”领域模型“的检验标准就是其是否能适应”问题域模型“的变化，其中”开闭原则“是在这里适用的。

“领域驱动设计”的第一层”驱动“的含义是”问题域模型”驱动”领域建模”以及”测试建模”。从这一点来说，”领域驱动设计”是适合一切场景的，”领域驱动设计“有问题的情况准确地来讲是目前的”领域建模技术”还不完善，当然也可以说是”领域建模技术”的运用有问题，“面向对象编程走错了路”并不是 OO 错了，而是“传统 OO”有很大的缺陷，首先第一条就是“没有面向消息的传递机制”。

领域建模的真谛是认清业务发展变化的脉络，其最高境界就是“庖丁解牛”，但是真实世界中并不存在“稳定领域”，业务的是永恒变化的。如果业务永远不变，那么只要满足用户需求，怎么设计都可以。

“领域驱动设计”的第二层含义是“驱动设计”。显然，“设计”只是一种手段和工具，是为“领域”服务的。“设计”的过程和质量必须通过“领域模型”来检验和验证，这也是测试驱动的本质。

尽可能不要让设计工具本身的缺陷扭曲了“领域”本身，否则就本末倒置了。例如，Banq 批判的“基于关系数据库的业务设计”以及坚持“域模型不要让技术污染“。

在生活中也是如此，例如在骑车或者开车出行时，用户心中都有一辆真正属于自己的车，至于这个车是不是有轮子等都并不重要，关键是能让自己舒适地出行。显然”舒适地出行“才是真正的业务领域，而”轮子之类的“只是设计。

领域驱动设计、敏捷工程以及测试驱动是浑然一体的，三者紧密结合，形成一套新的软件开发方式，而且已经开始普及和成熟，未来可能专门开发出专门的 MDD 工具或 DSL 来支持领域开发。

模型驱动架构、开发和工程学以及丰富的模型驱动开发环境——例如 NeXTStep——都是在 80 年代末出现的，现在模型已经无处不在，然而开发工具的数量还是很少，业界还在寻找让模型驱动开发成为主流的方法。

“领域驱动设计”=“问题域模型驱动领域建模”+“领域建模驱动软件实现”。

248.4 Business

在面对企业多业务集成（如 ERP、MES、HR、多项目、PDM、MRO）模式时，需要根据问题来考虑解决业务建模的问题，并且让设计模式和领域模型的思想融入到业务模型中。

业务建模应该回归自然，这样就可以把 SOA、ESB 等的意图理解为打造一个企业的“神经脉络”，“OO”可以理解为“神经元”，它们之间的通讯就是靠生物电脉冲，也就是消息驱动。如果在项目开发中感觉不自然的时候，那一定是模型出问题了。

大项目需要从整体思考并自上而下的进行大的业务划分，并确定不同业务领域的边界，领域驱动设计往往只是强调了业务的水平分割。

在开始阶段就对对象进行建模，然后再进行归类划分模块，并且把垂直分割完全等同于包的划分的做法是错误的，正确的做法应该是前期以确认领域边界功能为主，后期以确认领域内的对象模型为主。

上面提到的领域的切分就是不断对企业业务知识的学习和分析。当对一个业务认识不清的时候，最好的办法就是不同企业环境下去分析这个业务，那这个业务的所有发展变化就清楚了。

领域的边界就是服务，或者说是对外提供服务的唯一入口，领域服务和领域对象模型是一个业务领域的 2 个不同侧面。

- 领域服务强调是从外向内看，反映了“外部对业务领域的使用功能”；
- 领域对象模型强调业务领域就像一个独立的具有一定自主能力的生命体，反映了“业务领域的内部运行机制”。

领域对象模型的功能是不能对外暴露的，否则会造成外部对领域对象的耦合，而且域对象到处暴露可能导致整个系统形成了一个领域对象相互关联的蜘蛛网，一旦项目变大需要不同的部分独自发展时就无法进行切割。为了解决松耦合，DDD 还提出了专门的防腐层。

领域模型是不断变化的。比如说，以“添加一个 user 对象”这个功能为例，一开始业务很简单，认为放到 User 对象中即可，但是后来随着业务的复杂度提升，“添加一个 user 对象必须先审批”，最后业务更复杂了，“在不同的企业里这个审批的流程也不一样”。

这里，对于外部而言，如果一开始就调用的 user 域对象的方法就需要重新修改，因为后来增加的业务应该是 user 与其他对象协作一起完成的，因此不能放在 User 内部，不过放在服务里就没问题了，外界只关心提供必要的创建 user 的信息即可，不关心内部怎实现的，怎么变化的，以至于内部有没有 User 这个对象都没有关系。

注意，服务里的 addUser 和 User 里的 add 方法是同时存在的，但是二者的侧重点不同。其中，服务里的 addUser 是业务方法，而 User 的 add 方法仅仅是对象方法，前者调用后者。这就是封装，同时也支持了一定的柔性，越大的项目越需要注意区分。

至于什么样的对象适合作为聚合根的领域对象，一般而言，user 在大多数情况下都不是领域对象，而 customer 可以是。因为 user 是泛指，除了在 HR 中 user 可以是领域对象。

想象一下，在一个企业里有好几个不同的单业务系统，都是基于领域驱动设计的；在这些不同的业务场景下，user 都可以聚合根，这很正常，因为人可以干不同的事情，不过在把这些业务集成起来时就有问题了，因此作为聚合根的领域对象应该是业务场景类，而不应该是类似 user 的这样的类。

对于 User 作为另外一个域中的对象，在其他领域内如何引用的问题，其实大多数业务应用中只需要一个仅仅包含 UserId 和 UserName 属性的对象就足够了，所以 User 在其他域里是值对象，不需要引用像 rich domain 中的那样的 User。

248.5 Modeling

“一上来先识别类，然后就考虑怎么分配职责”的做法是一种本末倒置的僵化的静态建模。这种方式往往让人落入一种只注重“形”而忽略“意”（业务本来的面目）的怪圈。

建模分为 2 个阶段，第一个是“有机的业务建模”，第二个是“技术建模”。“业务建模”和“技术建模”的区别是：“业务建模”完全是业务语言，不含任何技术成分，比如“类、值对象和职责”的概念，在“技术建模”中才涉及那些概念。“业务建模”做好了，“技术建模”就成了自然而然的东西了。

“有机的业务建模”的一个非常重要的特征就是“有机”，就是强调业务模型中的每一个“业务主体”都是一个生命体。这些业务主体都有“生命特征”，在一定条件下受到外界的刺激就会发生一定的行为。这些业务主体构成了整个业务模型。当然，这

些业务主体的规模和层次不同：比较大的业务主体就是“领域主体”，他有明确的领域边界；而领域主体的对外界响应其实是通过内部的“业务核心主体”的协作来完成的，有的看见的见摸得着————具有业务实体的特点，有的比较抽象————控制规则和流程，当然有的“业务核心主体”可能是个多核细胞。“有机的业务建模”有一个非常重要的工作就是“检查业务模型的生命健康状况”。

需要注意的是，这个业务建模可能是通过不完善的用户需求建立起来的，所以需要构建多种场景通过不同的刺激去看这个领域主体内部的响应中是否有“异常情况”，如果有那就要修改模型。

“技术建模”应该“对内功能和对外功能要分开”，领域对象不建议同时承担对内和对外的功能，但是小项目除外。服务对外要封装和隐藏领域内部的东西，提供的是服务接口，同时服务要负责领域内部对象的行为组装。

从另外一个层面看，服务是需求，领域对象模型是实现。值对象的本质就是反映业务主体的不同业务特征，可能是业务主体的临时状态（比如用户的发帖数）或者属于另一业务主体的状态值。

- 从微观看，聚合是交互最紧密的业务对象的封装；
- 从宏观看，聚合是一个具有明确业务边界的独立业务核心主体。

聚合只是形式，业务模型的正确建立才是决定要素。

在实际建模过程中，从用户的角度看，业务本来就是面向关系和过程的；从设计看，业务是不同主体在相互作用，因此这也就是为什么越靠近用户的地方面向关系和过程的设计味道越浓的原因。

如果建模分成分析和设计两个阶段，也就是业务建模和技术建模，这样做会产生两者隔阂，业务建模注重如何反应需求，技术建模比较注重技术实现，两个关注点不一样，那么两者如何衔接就成了没人关注的，但是最重要的部分。

如果将两者合并在一起，只是按照 DDD 是不够的，DDD 只是一种静态建模，更多的是考虑特征特点等静态对象属性，而没有从行为职责去考虑，所以这时需要引入对象职责，通过对象职责和协作来完成聚合以及层次以及流程，确定聚合大边界和内部协作对象小边界，分离出规则和流程等。

另一方面，业务建模和技术建模的过程完全就是一体两面，而把它们分开是因为各种技术名词束缚了我们对业务的真正认识。

从“事件”角度看，“职责的本质就是事件的响应”，如果当职责分配不能合理地表示出来，就说明 OO 这方面到头了。其实就是 OO 这种世界观能把世界认识到什么一种程度而已。

过去的 OO 过于僵化、教条，偏重于静态场景，忽略了客观世界运动和相互作用的本质规律，而最近出现新技术和对 OO 的反思“正在回归自然”。例如，过去的业务模型更多表现在用例图、流程图和状态图等（与类无关的，是对业务的一种描述)。其中，用例图就是一种对边界捕捉的方法。

类的这些描述是静态的，所以只分析类图是不全面，或者说是危险的，所以出现了时序图和数据流图等来更好地、全面地描述业务。

即使理解了两面一体，问题仍然在于中间，中间怎样过渡很重要，而且即使掌握了业务模型也不知道怎样把这两面联系起来——也就是到底具体怎么表示。

建模是动态的、基于场景交互的，应该用更自然的还原业务本来面目的眼光去审视建模过程，也就是说“有机的业务建模”实际上就是“技术建模”的问题域建模，而“技术建模”只是“业务建模”的技术落地而已。这和过去的方式是完全不同的，因此一气呵成应该是领域驱动建模的一个特点，区别于过去方法的重要特征。

过早地进入“技术建模”或者“用技术建模来替代业务建模”是有危险的，容易犯“形而上学”的毛病，把“TDD 思想提前到业务建模阶段”对软件的质量非常重要，让技术建模更加容易。

企业业务建模的终极目标一定是非常复杂的（也就是我原来常说的大项目场景)，而且只有在复杂的场景下才能真正检验各种建模技术的偏颇。

从层次上来讲，企业的业务建模可以分为两个层面，”宏观建模”和“微观建模”。“宏观建模”是指首先要对企业做一个整体的信息化规划，对企业进行整体的的业务架构建模，其成果就是业务组件。其中的方法论可以参照 IBM 的 CBM，真正的落地还要靠自己对 CBM 的领悟。

DDD 主要属于“微观建模”部分，“微观建模”的两个方面——“结构化建模”和“行为建模”是一体两面。Evans 对 DDD 总结了关键的要素：实体、值对象、聚合、工厂和存储，不过其中还少了一个非常重要和关键的要素——“事件”。

众所周知，人体是由很多细胞构成的，那细胞之间是如何作用的呢，其实就是“刺激”和“响应”。其中“刺激”就是“事件”，所以“事件”是业务模型本来就应该具备的要素。

从这点来看，用户发帖这个动作激发一个“发帖事件”，这个事件会导致一系列的对象和值对象的状态变化。

- “结构化建模”是指建模中除了静态的实体和值对象的结构关系外，从“事件”角度看，实体或者值对象还具备一些“本能的反应”，比如” 手指会弯曲”。
- “行为建模”是指通过神经中枢（消息总线）来控制不同对象的本能反应来完成一个复杂的组合，比如” 用手弹钢琴”。

用事件来实现职责，在过去 SOA 以及传统架构比较常见，也有 Scala 的邮箱通讯机制，如果说事件机制类似观察者模式是一种分布式通讯，那么 DCI 则类似 Mediator 模式，它是一种封装通讯，其实比 Mediator 更彻底，根本就没有事件概念，直接和角色、职责、场景等业务建模概念结合。

目前的“领域建模”可能大多只停留在描述或者仿生“领域的事物 (what) 及活动 (how)”，可能并没有深入研究领域的模式或规律 (why)。

举个例子，复杂网络的建模，复杂网络的具体例子有生物网络、交通网络、互联网、神经网络、社会网络等，科学家要从这些网络抽象一个通用的模型，这个模型要足够简单，又不能太简单（比如，只将网络当成点与边的集合），一旦成功建模后，科学家们便可以研究模型找出规律，用于指导实践和预测未来。可是构造一个通用的模型几乎不可能，建立起来的模型往往只能反映现实的部分特性。

建模与需求分析的关系不能简单的认为是“需求分析的目标是就是建模，建模后指导技术实现”。例如，即时通信的领域模型本身也有助于我们做好需求分析，如果有可能，可以放在需求分析之前，或者对两者以拉锯式、互相推动的方式进行思考。

Part CVII

phpQuery

Chapter 249

Overview

phpQuery is a server-side, chainable, CSS3 selector driven Document Object Model (DOM) API based on jQuery JavaScript Library.

phpQuery (http://github.com/TobiaszCudnik/phpquery) 是一个服务器端运行的链式的、CSS3 选择器驱动的 DOM 解析工具。

phpQuery 参考了 jQuery 的语法，基于 PHP5 开发，并且提供了额外的命令行工具。

```
$ pear channel-discover phpquery-pear.appspot.com
$ pear install phpquery/phpQuery
```

使用 phpQuery 采集网页的示例如下：

```
$ phpquery 'http://code.google.com/p/phpquery/downloads/list?can=1' --find '.vt.col_4 a'
    --contents --getString null array_sum
```

phpQuery 依赖 PHP 的 DOM 扩展

```
<?php
require('phpQuery/phpQuery.php');
// INITIALIZE IT
// phpQuery::newDocumentHTML($markup);
// phpQuery::newDocumentXML();
// phpQuery::newDocumentFileXHTML('test.html');
// phpQuery::newDocumentFilePHP('test.php');
// phpQuery::newDocument('test.xml', 'application/rss+xml');
// this one defaults to text/html in utf8
$doc = phpQuery::newDocument('<div/>');
// FILL IT
// array syntax works like ->find() here
$doc['div']->append('<ul></ul>');
// array set changes inner html
$doc['div ul'] = '<li>1</li> <li>2</li> <li>3</li>';
// MANIPULATE IT
$li = null;
```

```
// almost everything can be a chain
$doc['ul > li']
  ->addClass('my-new-class')
  ->filter(':last')
    ->addClass('last-li')
// save it anywhere in the chain
    ->toReference($li);
// SELECT DOCUMENT
// pq(); is using selected document as default
phpQuery::selectDocument($doc);
// documents are selected when created or by above method
// query all unordered lists in last selected document
$ul = pq('ul')->insertAfter('div');
// ITERATE IT
// all direct LIs from $ul
foreach($ul['> li'] as $li) {
  // iteration returns PLAIN dom nodes, NOT phpQuery objects
  $tagName = $li->tagName;
  $childNodes = $li->childNodes;
  // so you NEED to wrap it within phpQuery, using pq();
  pq($li)->addClass('my-second-new-class');
}
// PRINT OUTPUT
// 1st way
print phpQuery::getDocument($doc->getDocumentID());
// 2nd way
print phpQuery::getDocument(pq('div')->getDocumentID());
// 3rd way
print pq('div')->getDocument();
// 4th way
print $doc->htmlOuter();
// 5th way
print $doc;
// another...
print $doc['ul'];
```

249.1 Basics

```
phpQuery::newDocumentFileXHTML('my-xhtml.html')->find('p'); $ul = pq('ul');
```

249.1.1 Loading Document

- `phpQuery::newDocument($html, $contentType = null)`
  Creates new document from markup. If no $contentType, autodetection is made (based on markup). If it fails, text/html in utf-8 is used.
- `phpQuery::newDocumentFile($file, $contentType = null)`
  Creates new document from file. Works like newDocument()
- `phpQuery::newDocumentHTML($html, $charset = 'utf-8')`
- `phpQuery::newDocumentXHTML($html, $charset = 'utf-8')`
- `phpQuery::newDocumentXML($html, $charset = 'utf-8')`
- `phpQuery::newDocumentPHP($html, $contentType = null)`
  Read more about it on PHPSupport page
- `phpQuery::newDocumentFileHTML($file, $charset = 'utf-8')`
- `phpQuery::newDocumentFileXHTML($file, $charset = 'utf-8')`
- `phpQuery::newDocumentFileXML($file, $charset = 'utf-8')`
- `phpQuery::newDocumentFilePHP($file, $contentType)`

249.1.2 pq() function

```
pq($param, $context = null);
```

pq() 类似于 jQuery 的 $()，pq() 支持三种类型的操作。

1. Importing markup

```
// Import into selected document:
// doesn't accept text nodes at beginning of input string
pq('<div/>')
// Import into document with ID from $pq->getDocumentID():
pq('<div/>', $pq->getDocumentID())
// Import into same document as DOMNode belongs to:
pq('<div/>', DOMNode)
// Import into document from phpQuery object:
pq('<div/>', $pq)
```

2. Running queries

```
// Run query on last selected document:
pq('div.myClass')
// Run query on document with ID from $pq->getDocumentID():
pq('div.myClass', $pq->getDocumentID())
// Run query on same document as DOMNode belongs to and use node(s)as root for
    query:
pq('div.myClass', DOMNode)
// Run query on document from phpQuery object
// and use object's stack as root node(s) for query:
pq('div.myClass', $pq)
```

3. Wrapping DOMNodes with phpQuery objects

```
foreach(pq('li') as $li) // $li is pure DOMNode, change it to phpQuery object pq(
    $li);
```

249.2 jQuery

phpQuery 提供了很多和 jQuery 类似的特性，例如：

- htmlOuter()
- xml()
- xmlOuter()
- markup()
- markupOuter()
- getString()
- reverse()
- contentsUnwrap()
- switchWith()
- all from PHPSupport
- all from Basic
- all from MultiDocumentSupport

249.2.1 Selectors

249.2.2 Attributes

249.2.3 Traversing

249.2.4 Manipulation

249.2.5 Ajax

249.2.6 Events

249.2.7 Utilities

249.2.8 Plugin

Chapter 250

QueryList

250.1 Overview

如果需要抓取一个网页的内容，但是只需要特定部分的信息，通常会用正则来解决。

虽然正则是一个通用解决方案，不过特定情况下，往往有更简单快捷的方法。例如，为了查询一个编程方面的问题，除了可以使用 Google，stackoverflow 作为一个专业的编程问答社区，会提供给你更多，更靠谱的答案。

不推荐在 HTML 页面使用正则，如果 HTML 页面过大，正则的效率无法保证，这种情况下可以考虑使用 phpQuery 来获取某个特定元素。

phpQuery 是一个用 php 实现的类似 jQuery 的开源项目，可以在服务器端以 jQuery 的语法形式解析网页元素。

phpQuery 支持使用类似 jQuery 的选择器，而且 phpQuery 可以对 DOM 进行任何复杂的操作。

QueryList 相当于 phpQuery 的子集，可以发挥 phpQuery 采集方面的强大功能。

作为一个基于 phpQuery 的 PHP 通用列表采集类，QueryList 让使用 PHP 采集网页信息时和使用 jQuery 选择元素一样简单。

- QueryList 只有一个核心的 API——静态方法 Query
- QueryList 支持使用 jQuery 选择器来选择页面元素
- QueryList 内置的过滤功能可以清洗无用的内容数据
- QueryList 支持扩展来实现多线程批量采集，模拟登陆采集等功能
- QueryList 采集结果直接以采集规则并按照列表的形式有序的返回
- QueryList 支持无限层级嵌套采集

QueryList3 使用 PSR-4 和 Composer，不兼容以前的版本。

```
$ composer require jaeger/querylist
```

在使用 Composer 安装了 QueryList 之后，只需要引入 vendor/autoload.php 文件就可以使用 QueryList 及其所有插件（如果安装了插件的话）。

```
<?php
require 'vendor/autoload.php';

use QL/QueryList;

$hj = QueryList::Query('http://mobile.csdn.net/', array("url"=>array('.unit h1 a', 'href
    ')));
```

```
$data = $hj->getData(function($x){
   return $x['url'];
});
print_r($data);
```

QueryList 扩展通过 QueryList::run() 方法运行。

QueryList 扩展可以根据需要选择性地安装：

- Request 网络操作扩展

```
$ composer require jaeger/querylist-ext-request
```

- Multi 多线程扩展

```
$ composer require jaeger/querylist-ext-multi
```

- Login 模拟登录扩展

```
$ composer require jaeger/querylist-ext-login
```

包含 QueyListy 以及扩展的完整配置如下：

*/,

```
{
    "require": {
        "jaeger/querylist": "^3.1",
        "jaeger/querylist-ext-request":"^1.0",
        "jaeger/querylist-ext-multi":"^1.0",
        "jaeger/querylist-ext-login":"^1.0"
    }
}
```

以下类库是上面扩展的依赖，安装扩展的时候会自动安装，也可以选择单独引入使用：

- Http 类

```
$ composer require jaeger/http
```

- CurlMulti 多线程类

```
$ composer require jaeger/curlmulti
```

250.1.1 Login

QueryList 的 Login 扩展可以实现模拟登陆然后采集。

250.1.2 Multi

QueryList 的 Multi 扩展支持多线程（多进程）采集。

250.1.3 Request

QueryList 的 Request 扩展可以实现携带 cookie、伪造来路等任意复杂的网络请求。