PHP最佳实践

极客学院出版

前言

PHP 是一门复杂的语言，经过多年折腾，使其不同版本之间高度不一致，有时还有些 bug。 每个版本都有自己独有的特性、多余和怪异之处，也很难跟踪哪个版本有哪些问题。 这也就很好理解为什么有时它会遭到那么多的厌恶。

尽管如此，如今它还是 Web 开发方面最流行的语言。 因其悠久的历史，对于实现密码哈希和数据库访问诸如此类的基本任务你能够找到很多教程。 但问题在于，5 个教程，你就很有可能找到 5 种完全不同的完成任务的方式，那么哪种是「正确」的方式呢？ 其他方式有难以捉摸的 bug 或者陷阱？ 确实很难搞明白，所以你经常要在互联网上反复查找尝试确认正确的答案。

这也是 PHP 编程新手频繁地因为丑陋、过时、或不安全的代码而遭到责备的原因之一。 如果 Google 搜索的第一个结果是一篇 4 年前的文章，讲述一种 5 年前的方法，那么 PHP 新手们也就很难改变经常遭受责备的现状。

本文档通过为 PHP 中常见的令人困惑的问题和任务编辑组织一系列被认为最佳实践的基本做法， 来尝试解决上述问题。 若一个低层次的任务在PHP中有多种令人困惑的实现方式，本文也会涵盖。

是什么

这是一份指南，在 PHP 程序员遇到一些常见低层次任务但不明确最佳做法（由于 PHP 可能提供了多种解决方案）之时，为其建议最佳实践。 例如：连接数据库是一个常见任务，PHP 中提供了大量可行的方案，但并不是所有的都是好的做法，因此，本文也会包含该问题。 本文包含的是一系列简短的、入门性质的方案。 涉及的示例在基本设定下就能够运行起来，你研究一下应该就能把它们变为对你有用的东西。 本文将指出一些我们认为是 PHP 中最新最好的东西。然而，这意味如果你在使用老版本的 PHP， 一些用来实现这些解决方案的特性对你并不可用。 这份文档会一直更新，我会尽我最大努力保持该文档与 PHP 的发展同步。

不是什么

本文档不是一份 PHP 教程。你应该在别处学习语言基础和语法。 它也不是一份针对 web 应用常见问题，如 cookie 存储、缓存、编程风格、文档等的指南。

它也不是一个安全指南。当本文档触碰到一些安全相关的问题时，也是希望你自己做些研究来确保你的 PHP 应用的安全问题。 你的代码造成的问题应该都是自己的过错。

该文档也并不是在主张一种特定的编程风格、模式或者框架。

也不是在主张一种特定的方式来完成高层次任务如用户注册、登录系统等。 本文档只限于 PHP 的悠久历史所造成的一些易混淆或不明确的低层次任务。

它不是一个一劳永逸的解决方案，也不是一个唯一的方案。 下面要讲述的一些方法对于你的特定场景来说也许并不是最好的，存在很多不同的方式来达到同样的目的。 特别是，高负载 web 应用也许能从更加难懂的方案中获益更多。

目录

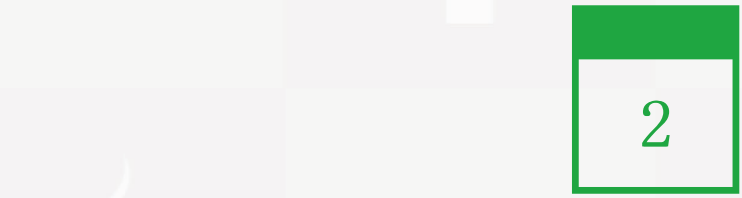
1

我们在使用哪个版本的 PHP？

带 Suhosin-Patch 的 PHP 5.3.10-1ubuntu3.6，安装在 Ubuntu 12.04 LTS 上。

PHP 是 Web 世界里的百年老龟，它的壳上铭刻着一段丰富、复杂、而粗糙的历史。 在一个共享主机的环境里，它的配置可能会限制你能做的事情。 为了保持清晰地叙述，我们将仅针对一个版本的 PHP 进行讲述。 在 2013 年 4 月 30 日时，该版本为 PHP 5.3.10-1ubuntu3.6 with Suhosin-Patch。 若你在 Ubuntu 12.04 LTS 服务器上使用 apt-get 进行安装的就是该版本的 PHP。

你也许发现这些方案中的一些在其他或者更老版本的 PHP 上也能工作。 如果是这样的话，就 由你来研究在这些更老版本上潜在的难以捉摸的 bug 或安全问题。



2

存储密码

使用 phpass 库来哈希和比较密码

经 phpass 0.3 测试，在存入数据库之前进行哈希保护用户密码的标准方式。 许多常用的哈希算法如 md5，甚至是 sha1 对于密码存储都是不安全的， 因为骇客能够使用那些算法轻而易举地破解密码。

对密码进行哈希最安全的方法是使用 bcrypt 算法。开源的 phpass 库以一个易于使用的类来提供该功能。

示例

```
<?php
// Include phpass 库
require_once('phpass-03/PasswordHash.php')

// 初始化散列器为不可移植(这样更安全)
$hasher = new PasswordHash(8, false);

// 计算密码的哈希值。$hashedPassword 是一个长度为 60 个字符的字符串.
$hashedPassword = $hasher->HashPassword('my super cool password');

// 你现在可以安全地将 $hashedPassword 保存到数据库中!

// 通过比较用户输入内容（产生的哈希值）和我们之前计算出的哈希值，来判断用户是否输入了正确的密码
$hasher->CheckPassword('the wrong password', $hashedPassword);  // false

$hasher->CheckPassword('my super cool password', $hashedPassword);  // true
?>
```

陷阱

许多资源可能推荐你在哈希之前对你的密码“加盐”。想法很好，但 phpass 在 HashPassword() 函数中已经对你的密码“加盐”了，这意味着你不需要自己“加盐”。

进一步阅读

- phpass
- 为什么使用 md5 或 sha 哈希密码是不安全的（中文）
- 怎样安全地存储密码

3

PHP 与 MySQL

使用 PDO 及其预处理语句功能。

在 PHP 中，有很多方式来连接到一个 MySQL 数据库。PDO（PHP 数据对象）是其中最新且最健壮的一种。 PDO 跨多种不同类型数据库有一个一致的接口，使用面向对象的方式，支持更多的新数据库支持的特性。

你应该使用 PDO 的预处理语句函数来帮助防范 SQL 注入攻击。 使用函数 bindValue 来确保你的 SQL 免于一级 SQL 注入攻击。 （虽然并不是 100% 安全的，查看进一步阅读获取更多细节。） 在以前，这必须使用一些「魔术引号(magic quotes)」函数的组合来实现。PDO 使得那堆东西不再需要。

示例

```
<?php
try{
    // 新建一个数据库连接
    // You'll probably want to replace hostname with localhost in the first parameter.
    // The PDO options we pass do the following:
    // \PDO::ATTR_ERRMODE enables exceptions for errors.  This is optional but can be handy.
    // \PDO::ATTR_PERSISTENT disables persistent connections, which can cause concurrency issues in certain cases.  See
    // \PDO::MYSQL_ATTR_INIT_COMMAND alerts the connection that we'll be passing UTF-8 data.
    // This may not be required depending on your configuration, but it'll save you headaches down the road
    // if you're trying to store Unicode strings in your database.  See "Gotchas".
    $link = new \PDO(   'mysql:host=your-hostname;dbname=your-db',
                        'your-username',
                        'your-password',
                        array(
                            \PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION,
                            \PDO::ATTR_PERSISTENT => false,
                            \PDO::MYSQL_ATTR_INIT_COMMAND => 'set names utf8mb4'
                        )
                    );

    $handle = $link->prepare('select Username from Users where UserId = ? or Username = ? limit ?');

    // PHP bug: if you don't specify PDO::PARAM_INT, PDO may enclose the argument in quotes.
    // This can mess up some MySQL queries that don't expect integers to be quoted.
    // See: https://bugs.php.net/bug.php?id=44639
    // If you're not sure whether the value you're passing is an integer, use the is_int() function.
    $handle->bindValue(1, 100, PDO::PARAM_INT);
    $handle->bindValue(2, 'Bilbo Baggins');
    $handle->bindValue(3, 5, PDO::PARAM_INT);

    $handle->execute();

    // Using the fetchAll() method might be too resource-heavy if you're selecting a truly massive amount of rows.
    // If that's the case, you can use the fetch() method and loop through each result row one by one.
    // You can also return arrays and other things instead of objects.  See the PDO documentation for details.
    $result = $handle->fetchAll(\PDO::FETCH_OBJ);

    foreach($result as $row){
        print($row->Username);
    }
}
```

```
catch(\PDOException $ex){
    print($ex->getMessage());
}
?>
```

陷阱

- 当绑定整型变量时，如果不传递 PDO::PARAM_INT 参数有事可能会导致 PDO 对数据加引号。 这会搞坏特定的 MySQL 查询。查看该 bug 报告。
- 未使用 `set names utf8mb4` 作为首个查询，可能会导致 Unicode 数据错误地存储进数据库，这依赖于你的配置。 如果你绝对有把握你的 Unicode 编码数据不会出问题，那你可以不管这个。
- 启用持久连接可能会导致怪异的并发相关的问题。 这不是一个 PHP 的问题，而是一个应用层面的问题。只要你仔细考虑了后果，持久连接一般会是安全的。 查看 Stack Overfilow 这个问题。
- 即使你使用了 `set names utf8mb4`，你也得确认实际的数据库表使用的是 utf8mb4 字符集！
- 可以在单个 execute() 调用中执行多条 SQL 语句。 只需使用分号分隔语句，但注意这个 bug，在该文档所针对的 PHP 版本中还没修复。
- Laruence：PDOStatement::bindParam 的一个陷阱

进一步阅读

- PHP 手册：PDO
- 为什么你应该使用 PHP 的 PDO 访问数据库 （中文）
- Stack Overflow: PHP PDO vs 普通的 mysql_connect
- Stack Overflow: PDO 预处理语句足以防范 SQL 注入吗？
- Stack Overflow: 在 MySQL 中使用 SET NAMES utf8？

4

PHP 标签

使用 。

有几种不同的方式用来区分 PHP 程序块：，，，以及<% %>。 对于打字来说，更短的标签更方便些，但唯一一种在所有 PHP 服务器上都一定能工作的标签是。 若你计划将你的 PHP 应用部署到一台上面的 PHP 配置你无法控制的服务器上，那么你应始终使用 。

若你仅仅是为自己编码，也能控制你将使用的 PHP 配置，你可能觉得短标签更方便些。 但记住 可能会和 XML 声明冲突，并且实际上是 ASP 的风格。

无论你选择哪一种，确保一致。

陷阱

- 在一个纯 PHP 文件（例如，仅包含一个类定义的文件）中包含一个关闭?>标签时，确保其后不会跟着任何换行。 当 PHP 解析器安全地吃进跟在关闭标签之后的单个换行符时，任何其他的换行都可能被输出到浏览器，如果之后要输出某些 HTTP 头，那么可能会造成混淆。
- 编写Web应用时，确保在关闭?>标签与 html 的标签之间不会留下换行。 正确的 HTML 文件中，标签必须是文件中的第一样东西---在其之前的任何空格或换行都会使其无效。

进一步阅读

- Stack Overflow: 可以使用 PHP 短标签吗?

5

自动加载类

使用 spl_autoload_register() 来注册你的自动加载函数。

PHP 提供了若干方式来自动加载包含还未加载的类的文件。 老的方法是使用名为 __autoload() 魔术全局函数。然而你一次仅能定义一个 __autoload() 函数，因此如果你的程序包含一个也使用了 __autoload() 函数的库，就会发生冲突。

处理这个问题的正确方法是唯一地命名你的自动加载函数，然后使用 spl_autoload_register() 函数来注册它。该函数允许定义多个 __autoload() 这样的函数，因此你不必担心其他代码的 __autoload() 函数。

示例

```
<?php
// 首先，定义你的自动载入的函数
function MyAutoload($className){
    include_once($className . '.php');
}

// 然后注册它。
spl_autoload_register('MyAutoload');

// Try it out!
// 因为我们没包含一个定义有 MyClass 的文件，所以自动加载器会介入并包含 MyClass.php。
// 在本例中，假定在 MyClass.php 文件中定义了 MyClass 类。
$var = new MyClass();
?>
```

进一步阅读

- PHP手册：spl_autoload_register()
- Stack Overflow: 高效的 PHP 自动加载和命名策略

6

从性能角度来看单引号和双引号

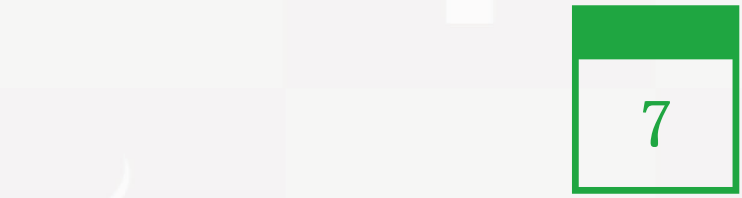
其实并不重要。

已有很多人花费很多笔墨来讨论是使用单引号（'）还是双引号（"）来定义字符串。 单引号字符串不会被解析，因此放入字符串的任何东西都会以原样显示。 双引号字符串会被解析，字符串中的任何 PHP 变量都会被求值。 另外，转义字符如换行符 \n 和制表符 \t 在单引号字符串中不会被求值，但在双引号字符串中会被求值。

由于双引号字符串在程序运行时要求值，从而理论上使用单引号字符串能提高性能，因为 PHP 不会对单引号字符串求值。 这对于一定规模的应用来说也许确实如此，但对于现实中一般的应用来说， 区别非常小以至于根本不用在意。因此对于普通应用，你选择哪种字符串并不重要。 对于负载极其高的应用来说，是有点作用的。 根据你的应用的需要来做选择，但无论你选择什么，请保持一致。

进一步阅读

- PHP 手册：字符串
- PHP 基准（向下滚动到引号类型(Quote Types)）
- Stack Overflow: PHP 中单引号字符串相比双引号字符串有性能优势么？
- Laruence：PHP的单引号和双引号



define() vs. const

使用 define()，除非考虑到可读性、类常量、或关注微优化

习惯上，在 PHP 中是使用 define() 函数来定义常量。 但从某个时候开始，PHP 中也能够使用 const 关键字来声明常量了。 那么当定义常量时，该使用哪种方式呢？

答案在于这两种方法之间的区别。

1. define() 在执行期定义常量，而 const 在编译期定义常量。这样 const 就有轻微的速度优势， 但不值得考虑这个问题，除非你在构建大规模的软件。
2. define() 将常量放入全局作用域，虽然你可以在常量名中包含命名空间。 这意味着你不能使用 define() 定义类常量。
3. define() 允许你在常量名和常量值中使用表达式，而 const 则都不允许。 这使得 define() 更加灵活。
4. define() 可以在 if() 代码块中调用，但 const 不行。

示例

```
<?php
// 来看看这两种方法如何处理 namespaces
namespace MiddleEarth\Creatures\Dwarves;
const GIMLI_ID = 1;
define('MiddleEarth\Creatures\Elves\LEGOLAS_ID', 2);

echo(\MiddleEarth\Creatures\Dwarves\GIMLI_ID);  // 1
echo(\MiddleEarth\Creatures\Elves\LEGOLAS_ID);  // 2; 注意：我们使用了 define()

// Now let's declare some bit-shifted constants representing ways to enter Mordor.
define('TRANSPORT_METHOD_SNEAKING', 1 << 0); // OK!
const TRANSPORT_METHOD_WALKING = 1 << 1; //Compile error! const can't use expressions as values

// 接下来, 条件常量。
define('HOBBITS_FRODO_ID', 1);

if($isGoingToMordor){
    define('TRANSPORT_METHOD', TRANSPORT_METHOD_SNEAKING); // OK!
    const PARTY_LEADER_ID = HOBBITS_FRODO_ID // 编译错误: const 不能用于 if 块中
}

// 最后, 类常量
class OneRing{
    const MELTING_POINT_DEGREES = 1000000; // OK!
    define('SHOW_ELVISH_DEGREES', 200); // 编译错误: 在类内不能使用 define()
}
?>
```

小插曲：当我看到第一行的 MiddleEarth 还没有感觉到什么，再往下看到 Mordor 时，震惊了。`OneRing`，`One Ring`，`OneRinggggggg!`

因为 define() 更加灵活，你应该使用它以避免一些令人头疼的事情，除非你明确地需要类常量。 使用 const 通常会产生更加可读的代码，但是以牺牲灵活性为代价的。

无论你选择哪一种，请保持一致。

进一步阅读

- Stack Overflow: define() vs. const
- PHP 手册：常量
- Stack Overflow: define() vs. 变量

8

缓存 PHP opcode

使用 APC

在一个标准的 PHP 环境中，每次访问PHP脚本时，脚本都会被编译然后执行。 一次又一次地花费时间编译相同的脚本对于大型站点会造成性能问题。

解决方案是采用一个 opcode 缓存。 opcode 缓存是一个能够记下每个脚本经过编译的版本，这样服务器就不需要浪费时间一次又一次地编译了。 通常这些 opcode 缓存系统也能智能地检测到一个脚本是否发生改变，因此当你升级 PHP 源码时，并不需要手动清空缓存。

PHP 5.5 内建了一个缓存 OPcache。PHP 5.2 - 5.4 下可以作为PECL扩展安装。

此外还有几个PHP opcode 缓存值得关注 eaccelerator， xcache，以及APC。 APC 是 PHP 项目官方支持的，最为活跃，也最容易安装。 它也提供一个可选的类 memcached 的持久化键-值对存储，因此你应使用它。

安装 APC

在 Ubuntu 12.04 上你可以通过在终端中执行以下命令来安装 APC：

```
user@localhost: sudo apt-get install php-apc
```

除此之外，不需要进一步的配置。

将 APC 作为一个持久化键-值存储系统来使用

APC 也提供了对于你的脚本透明的类似于 memcached 的功能。 与使用 memcached 相比一个大的优势是 APC 是集成到 PHP 核心的，因此你不需要在服务器上维护另一个运行的部件， 并且 PHP 开发者在 APC 上的工作很活跃。 但从另一方面来说，APC 并不是一个分布式缓存，如果你需要这个特性，你就必须使用 memcached 了。

示例

```
<?php
// Store some values in the APC cache.  We can optionally pass a time-to-live,
// but in this example the values will live forever until they're garbage-collected by APC.
apc_store('username-1532', 'Frodo Baggins');
apc_store('username-958', 'Aragorn');
apc_store('username-6389', 'Gandalf');

// After storing these values, any PHP script can access them, no matter when it's run!
$value = apc_fetch('username-958', $success);
if($success === true)
    print($value); // Aragorn

$value = apc_fetch('username-1', $success); // $success will be set to boolean false, because this key doesn't exist.
if($success !== true) // Note the !==, this checks for true boolean false, not "falsey" values like 0 or empty string.
    print('Key not found');

apc_delete('username-958'); // This key will no longer be available.
?>
```

陷阱

如果你使用的不是 PHP-FPM（例如你在使用 mod_php 或 mod_fastcgi）， 那么每个 PHP 进程都会有自己独有的 APC 实例，包括键-值存储。 若你不注意，这可能会在你的应用代码中造成同步问题。

进一步阅读

- PHP 手册：APC
- Laruence：深入理解 PHP 原理之 Opcodes

9

PHP 与 Memcached

若你需要一个分布式缓存，那就使用 Memcached 客户端库。否则，使用 APC。

缓存系统通常能够提升应用的性能。Memcached 是一个受欢迎的选择，它能配合许多语言使用，包括 PHP。

然而，从一个 PHP 脚本中访问一个 Memcached 服务器，你有两个不同且命名很愚蠢的客户端库选择项： Memcache 和 Memcached。 它们是两个名字几乎相同的不同库，两者都可用于访问一个 Memcached 实例。

事实证明，Memcached 库对于 Memcached 协议的实现最好，包含了一些 Mmecache 库没有的有用的特性， 并且看起来 Memcached 库的开发也最为活跃。

然而，如果不需要访问来自一组分布式服务器的一个 Memcached 实例，那就使用 APC。 APC 得到 PHP 项目的支持，具备很多和 Memcached 相同的功能，并且能够用作 opcode 缓存，这能提高 PHP 脚本的性能。

安装M emcached 客户端库

在安装 Memcached 服务器之后，需要安装 Memcached 客户端库。没有该库，PHP 脚本就没法与 Memcached 服务器通信。

在 Ubuntu 12.04 上，你可以使用如下命令来安装 Memcached 客户端库：

```
user@localhost: sudo apt-get install php5-memcached
```

使用 APC 作为替代

查看 opcode 缓存一节阅读更多与使用 APC 作为 Memcached 替代方案相关的信息。

进一步阅读

- PHP 手册：Memcached
- PHP 手册：APC
- Stack Overflow: PHP 中使用 Memcache vs. Memcached
- Stack Overflow: Memcached vs APC，我该选择哪一个？

10

PHP 与正则表达式

使用 PCRE(preg_*)家族函数

PHP有两种使用不同的方式来使用正则表达式：PCRE（Perl兼容表示法，`preg_*`）函数 和 POSIX（POSIX 扩展表示法，`ereg_*`） 函数。

每个函数家族各自使用一种风格稍微不同的正则表达式。幸运的是，POSIX 家族函数从 PHP 5.3.0 开始就被弃用了。因此，你绝不应该使用 POSIX 家族函数编写新的代码。 始终使用 PRCE 家族函数，即 `preg_*` 函数。

进一步阅读

- PHP 手册：PCRE
- Laruence：PHP 正则之递归匹配
- PHP 正则表达式起步 （这么好的文章，咋就没有中文版呢，如果你发现了或者翻译了，请通知我）

11

配置 Web 服务器提供 PHP 服务

使用 PHP-FPM

有多种方式来配置一个 web 服务器以提供 PHP 服务。传统（并且糟糕的）的方式是使用 Apache 的 mod_php。Mod_php将PHP 绑定到 Apache 自身，但是 Apache 对于该模块功能的管理工作非常糟糕。一旦遇到较大的流量，就会遭受严重的内存问题。

后来两个新的可选项很快流行起来：mod_fastcgi 和 mod_fcgid。 两者均保持一定数量的 PHP 执行进程， Apache 将请求发送到这些端口来处理 PHP 的执行。由于这些库限制了存活的 PHP 进程的数量， 从而大大减少了内存使用而没有影响性能。

一些聪明的人创建一个 fastcgi 的实现，专门为真正与 PHP 工作良好而设计，他们称之为 PHP-FPM。PHP 5.3.0 之前，为安装它， 你得跨越许多障碍，但幸运的是，PHP 5.3.3 的核心包含了 PHP-FPM，因此在 Ubuntu 12.04 上安装它非常方便。

如下示例是针对 Apache 2.2.22 的，但 PHP-FPM 也能用于其他 web 服务器如 Nginx。

安装 PHP-FPM 和 Apache

在 Ubuntu 12.04 上你可以使用如下命令安装 PHP-FPM 和 Apache：

```
user@localhost: sudo apt-get install apache2-mpm-worker
libapache2-mod-fastcgi php5-fpm
user@localhost: sudo a2enmod actions alias fastcgi
```

注意我们 必须 使用 apache2-mpm-worker，而不是 apache2-mpm-prefork 或 apache2-mpm-threaded。 接下来配置 Aapache 虚拟主机将 PHP 请求路由到 PHP-FPM 进程。将如下配置语句放入 Apache 配置文件（在 Ubuntu 12.04 上默认配置文件是 /etc/apache2/sites-available/default）。

```
<VirtualHost *:80>
    AddHandler php5-fcgi .php
    Action php5-fcgi /php5-fcgi
    Alias /php5-fcgi /usr/lib/cgi-bin/php5-fcgi
    FastCgiExternalServer /usr/lib/cgi-bin/php5-fcgi -host 127.0.0.1:9000 -idle-timeout 120 -pass-header Authorization
</VirtualHost>
```

最后，重启 Apache 和 FPM 进程：

```
user@localhost: sudo service apache2 restart && sudo service php5-fpm
restart
```

进一步阅读

- PHP 手册：PHP-FPM
- PHP-FPM 主页
- 在 Ubuntu 服务器 Maverick 上安装 Apache + mod_fastcgi + PHP-FPM
- 为什么 mod_php 的性能很糟糕

12

发送邮件

使用PHPMailer

经 PHPMailer 5.1 测试

PHP 提供了一个 mail() 函数，看起来很简单易用。 不幸的是，与 PHP 中的很多东西一样，它的简单性是个幻象，因其虚假的表面使用它会导致严重的安全问题。

Email 是一组网络协议，比 PHP 的历史还曲折。完全可以说发送邮件中的陷阱与 PHP 的 mail() 函数一样多，这个可能会令你有点「不寒而栗」吧。

PHPMailer 是一个流行而成熟的开源库，为安全地发送邮件提供一个易用的接口。 它关注可能陷阱，这样你可以专注于更重要的事情。

示例

```
<?php
// Include the PHPMailer library
require_once('phpmailer-5.1/class.phpmailer.php');

// Passing 'true' enables exceptions.  This is optional and defaults to false.
$mailer = new PHPMailer(true);

// Send a mail from Bilbo Baggins to Gandalf the Grey

// Set up to, from, and the message body.  The body doesn't have to be HTML;
// check the PHPMailer documentation for details.
$mailer->Sender = 'bbaggins@example.com';
$mailer->AddReplyTo('bbaggins@example.com', 'Bilbo Baggins');
$mailer->SetFrom('bbaggins@example.com', 'Bilbo Baggins');
$mailer->AddAddress('gandalf@example.com');
$mailer->Subject = 'The finest weed in the South Farthing';
$mailer->MsgHTML('<p>You really must try it, Gandalf!</p><p>-Bilbo</p>');

// Set up our connection information.
$mailer->IsSMTP();
$mailer->SMTPAuth = true;
$mailer->SMTPSecure = 'ssl';
$mailer->Port = 465;
$mailer->Host = 'my smpt host';
$mailer->Username = 'my smtp username';
$mailer->Password = 'my smtp password';

// All done!
$mailer->Send();
?>
```

13

验证邮件地址

使用 filter_var() 函数

Web 应用可能需要做的一件常见任务是检测用户是否输入了一个有效的邮件地址。毫无疑问你可以在网上找到一些声称可以解决该问题的复杂的正则表达式，但是最简单的方法是使用 PHP 的内建 filter_val() 函数。

示例

```
<?php
filter_var('sgamgee@example.com', FILTER_VALIDATE_EMAIL);
//Returns "sgamgee@example.com". This is a valid email address.

filter_var('sauron@mordor', FILTER_VALIDATE_EMAIL);
// Returns boolean false! This is *not* a valid email address.
?>
```

进一步阅读

- PHP 手册：filter_var()
- PHP 手册：过滤器的类型

注意

邮件地址验证也可以交给前端解决。HTML 5 的 表单即支持验证邮箱地址。只需将input元素的type设为email，就会自动验证用户输入的是否是合法的邮件地址。

```
<input type="email" name="email"></pre>
```

14

净化 HTML 输入和输出

对于简单的数据净化，使用 htmlentities() 函数，复杂的数据净化则使用 HTML Purifier 库

经 HTML Purifier 4.4.0 测试

在任何 wbe 应用中展示用户输出时，首先对其进行“净化”去除任何潜在危险的 HTML 是非常必要的。 一个恶意的用户可以制作某些 HTML，若被你的 web 应用直接输出，对查看它的人来说会很危险。

虽然可以尝试使用正则表达式来净化 HTML，但不要这样做。HTML是一种复杂的语言，试图使用正则表达式来净化 HTML 几乎总是失败的。

你可能会找到建议你使用 strip_tags() 函数的观点。 虽然 strip_tags() 从技术上来说是安全的，但如果输入的不合法的 HTML（比如， 没有结束标签），它就成了一个「愚蠢」的函数，可能会去除比你期望的更多的内容。 由于非技术用户在通信中经常使用 < 和 > 字符，strip_tags() 也就不是一个好的选择了。

如果阅读了验证邮件地址一节， 你也许也会考虑使用 filter_var() 函数。 然而 filter_var() 函数在遇到断行时会出现问题， 并且需要不直观的配置以接近 htmlentities() 函数的效果， 因此也不是一个好的选择。

对于简单需求的净化

如果你的 web 应用仅需要完全地转义（因此可以无害地呈现，但不是完全去除） HTML， 则使用 PHP 的内建 htmlentities() 函数。 这个函数要比 HTML Purifier 快得多，因此它不对 HTML 做任何验证---仅转义所有东西。

htmlentities() 不同于类似功能的函数htmlspecialchars()， 它会编码所有适用的 HTML 实体，而不仅仅是一个小的子集。

示例

```
<?php
// Oh no!  The user has submitted malicious HTML, and we have to display it in our web app!
$evilHtml = '<div onclick="xss();">Mua-ha-ha!  Twiddling my evil mustache...</div>';

// Use the ENT_QUOTES flag to make sure both single and double quotes are escaped.
// Use the UTF-8 character encoding if you've stored the text as UTF-8 (as you should have).
// See the UTF-8 section in this document for more details.
$safeHtml = htmlentities($evilHtml, ENT_QUOTES, 'UTF-8');
// $safeHtml is now fully escaped HTML.  You can output $safeHtml to your users without fear!
?>
```

对于复杂需求的净化

对于很多 web 应用来说，简单地转义 HTML 是不够的。 你可能想完全去除任何HTML，或者允许一小部分子集的 HTML 存在。 若是如此，则使用 HTML Purifier 库。

HTML Purifier 是一个经过充分测试但效率比较低的库。 这就是为什么如果你的需求并不复杂就应使用 htmlentities()， 因为它的效率要快得多。

HTML Purifier 相比 strip_tags() 是有优势的， 因为它在净化 HTML 之前会对其校验。 这意味着如果用户输入无效 HTML，HTML Purifier 相比 strip_tags() 更能保留 HTML 的原意。 HTML Purifier 高度可定制，允许你为 HTML 的一个子集建立白名单来允许这个 HTML 子集的实体存在输出中。

但其缺点就是相当的慢，它要求一些设置，在一个共享主机的环境里可能是不可行的。 其文档通常也复杂而不易理解。 以下示例是一个基本的使用配置。 查看文档阅读 HTML Purifier 提供的更多更高级的特性。

示例

```
<?php
// Include the HTML Purifier library
require_once('htmlpurifier-4.4.0/HTMLPurifier.auto.php');

// Oh no!  The user has submitted malicious HTML, and we have to display it in our web app!
$evilHtml = '<div onclick="xss();">Mua-ha-ha!  Twiddling my evil mustache...</div>';

// Set up the HTML Purifier object with the default configuration.
$purifier = new HTMLPurifier(HTMLPurifier_Config::createDefault());

$safeHtml = $purifier->purify($evilHtml);
// $safeHtml is now sanitized.  You can output $safeHtml to your users without fear!
?>
```

陷阱

- 以错误的字符编码使用 htmlentities() 会造成意想不到的输出。 在调用该函数时始终确认指定了一种字符编码，并且该编码与将被净化的字符串的编码相匹配。 更多细节请查看 UTF-8 一节。
- 使用 htmlentities() 时，始终包含 ENT_QUOTES 和字符编码参数。 默认情况下，htmlentities() 不会对单引号编码。多愚蠢的默认做法！
- HTML Purifier 对于复杂的 HTML 效率极其的低。可以考虑设置一个缓存方案如APC来保存经过净化的结果以备后用。

进一步阅读

- PHP HTML 净化工具对比（英文）
- Laruence：PHP Taint - 一个用来检测 XSS/SQL/Shell 注入漏洞的扩展
- Stack Overflow: 使用 strip_tags() 来防止 XSS？
- Stack Overflow: PHP中净化用户输入的最佳方法是什么？
- Stack Overflow: 断行时的 FILTER_SANITIZE_SPECIAL_CHARS 问题

15

PHP 与 UTF-8

没有一行式解决方案。小心、注意细节，以及一致性。

PHP 中的 UTF-8 糟透了。原谅我的用词。

目前 PHP 在低层次上还不支持 Unicode。有几种方式可以确保 UTF-8 字符串能够被正确处理， 但并不容易，需要深入到 web 应用的所有层面，从 HTML，到 SQL，到 PHP。我们旨在提供一个简洁、 实用的概述。

PHP 层面的 UTF-8

基本的字符串操作，如串接 两个字符串、将字符串赋给变量，并不需要任何针对 UTF-8 的特殊东西。 然而，多数 字符串函数，如 strpos() 和 strlen，就需要特殊的考虑。 这些函数都有一个对应的 mb_* 函数：例如，mb_strpos() 和 mb_strlen()。 这些对应的函数统称为多字节字符串函数。 这些多字节字符串函数是专门为操作 Unicode 字符串而设计的。

当你操作 Unicode 字符串时，必须使用 mb_* 函数。 例如，如果你使用 substr() 操作一个 UTF-8 字符串，其结果就很可能包含一些乱码。 正确的函数应该是对应的多字节函数， mb_substr()。

难的是始终记得使用 mb_* 函数。即使你仅一次忘了，你的 Unicode 字符串在接下来的处理中就可能产生乱码。

并不是所有的字符串函数都有一个对应的 mb_*。如果不存在你想要的那一个，那你就只能自认倒霉了。

此外，在每个 PHP 脚本的顶部（或者在全局包含脚本的顶部）你都应使用 mb_internal_encoding 函数，如果你的脚本会输出到浏览器，那么还得紧跟其后加个mb_http_output() 函数。在每个脚本中显式地定义字符串的编码在以后能为你减少很多令人头疼的事情。

最后，许多操作字符串的 PHP 函数都有一个可选参数让你指定字符编码。 若有该选项， 你应始终显式地指明 UTF-8 编码。 例如，htmlentities() 就有一个字符编码方式选项，在处理这样的字符串时应始终指定 UTF-8。

MySQL 层面的 UTF-8

如果你的 PHP 脚本会访问 MySQL，即使你遵从了前述的注意事项，你的字符串也有可能在数据库中存储为非 UTF-8 字符串。

确保从 PHP 到 MySQL 的字符串为 UTF-8 编码的，确保你的数据库以及数据表均设置为 utf8mb4 字符集， 并且在你的数据库中执行任何其他查询之前先执行 MySQL 查询 `set names utf8mb4` 。这是至关重要的。 示例请查看连接并查询 MySQL 数据库一节内容。

注意你必须使用 `utf8mb4` 字符集来获得完整的 UTF-8 支持，而不是 `utf8` 字符集！原因请查看进一步阅读。

浏览器层面的 UTF-8

使用 mb_http_output() 函数 来确保你的 PHP 脚本输出 UTF-8 字符串到浏览器。 并且在 HTML 页面的 标签块中包含 字符集 标签块。

示例

```
<?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 = 'Aš galiu valgyti stikl? ir jis man?s ne?eid?ia';

// Transform the string in some way with a multibyte function
$string = mb_substr($string, 0, 10);

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

// Store our transformed string as UTF-8 in our database
// Assume our DB and tables are in the utf8mb4 character set and collation
$handle = $link->prepare('insert into Sentences (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 Sentences 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);
?><!doctype html>
```

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; 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>
```

进一步阅读

- PHP 手册：多字节字符串函数
- PHP UTF-8 备忘单
- Stack Overflow: 什么因素致使 PHP 不兼容 Unicode?
- Stack Overflow: PHP 与 MySQL 之间国际化字符串的最佳实践
- 怎样在MySQL数据库中完整支持Unicode

16

处理日期和时间

使用DateTime 类。

在 PHP 糟糕的老时光里，我们必须使用 date()， gmdate()， date_timezone_set()， strtotime()等等令人迷惑的 组合来处理日期和时间。悲哀的是现在你仍旧会找到很多在线教程在讲述这些不易使用的老式函数。

幸运的是，我们正在讨论的 PHP 版本包含友好得多的 DateTime 类。 该类封装了老式日期函数所有功能，甚至更多，在一个易于使用的类中，并且使得时区转换更加容易。 在PHP中始终使用 DateTime 类来创建，比较，改变以及展示日期。

示例

```
<?php
// Construct a new UTC date.  Always specify UTC unless you really know what you're doing!
$date = new DateTime('2011-05-04 05:00:00', new DateTimeZone('UTC'));

// Add ten days to our initial date
$date->add(new DateInterval('P10D'));

echo($date->format('Y-m-d h:i:s')); // 2011-05-14 05:00:00

// Sadly we don't have a Middle Earth timezone
// Convert our UTC date to the PST (or PDT, depending) time zone
$date->setTimezone(new DateTimeZone('America/Los_Angeles'));

// Note that if you run this line yourself, it might differ by an hour depending on daylight savings
echo($date->format('Y-m-d h:i:s')); // 2011-05-13 10:00:00

$later = new DateTime('2012-05-20', new DateTimeZone('UTC'));

// Compare two dates
if($date < $later)
    echo('Yup, you can compare dates using these easy operators!');

// Find the difference between two dates
$difference = $date->diff($later);

echo('The 2nd date is ' . $difference['days'] . ' later than 1st date.');
?>
```

陷阱

- 如果你不指定一个时区，DateTime::__construct() 就会将生成日期的时区设置为正在运行的计算机的时区。之后，这会导致大量令人头疼的事情。 在创建新日期时始终指定 UTC 时区，除非你确实清楚自己在做的事情。
- 如果你在 DateTime::__construct() 中使用 Unix 时间戳，那么时区将始终设置为 UTC 而不管第二个参数你指定了什么。
- 向 DateTime::__construct() 传递零值日期（如：“0000-00-00”，常见 MySQL 生成该值作为 DateTime 类型数据列的默认值）会产生一个无意义的日期，而不是“0000-00-00”。
- 在 32 位系统上使用 DateTime::getTimestamp() 不会产生代表 2038 年之后日期的时间戳。64 位系统则没有问题。

进一步阅读

- PHP 手册：DateTime 类
- Stack Overflow: 访问超出 2038 的日期

17

检测一个值是否为 null 或 false

使用 === 操作符来检测 null 和布尔 false 值。

PHP 宽松的类型系统提供了许多不同的方法来检测一个变量的值。 然而这也造成了很多问题。 使用 == 来检测一个值是否为 null 或 false，如果该值实际上是一个空字符串或 0，也会误报为 false。 isset 是检测一个变量是否有值， 而不是检测该值是否为 null 或 false，因此在这里使用是不恰当的。

is_null() 函数能准确地检测一个值是否为 null， is_bool 可以检测一个值是否是布尔值（比如 false）， 但存在一个更好的选择：=== 操作符。=== 检测两个值是否同一， 这不同于 PHP 宽松类型世界里的 相等。它也比 is_null() 和 is_bool() 要快一些，并且有些人认为这比使用函数来做比较更干净些。

示例

```
<?php
$x = 0;
$y = null;

// Is $x null?
if($x == null)
    print('Oops! $x is 0, not null!');

// Is $y null?
if(is_null($y))
    print('Great, but could be faster.');

if($y === null)
    print('Perfect!');

// Does the string abc contain the character a?
if(strpos('abc', 'a'))
    // GOTCHA!  strpos returns 0, indicating it wishes to return the position of the first character.
    // But PHP interpretes 0 as false, so we never reach this print statement!
    print('Found it!');

//Solution: use !== (the opposite of ===) to see if strpos() returns 0, or boolean false.
if(strpos('abc', 'a') !== false)
    print('Found it for real this time!');
?>
```

陷阱

- 测试一个返回 0 或布尔 false 的函数的返回值时，如 strpos()，始终使用 === 和!==，否则你就会碰到问题。

进一步阅读

- PHP 手册：比较操作符
- Stack Overflow: is_null() vs ===
- Laruence：isset 和 is_null 的不同