Bài viết này dành cho những lập trình viên PHP quan tâm đến việc tránh những lỗi thường gặp khi sử dụng PHP. Người đọc phải biết các cú pháp của PHP, và nên biết công dụng của các hàm trong PHP.

Một trong những điểm mạnh nhất của PHP vô tình trở thành một trong những điểm yếu nhất của nó: **tính dễ sử dụng**. Nhiều người chọn PHP vì tính dễ sử dụng, đã không nhận ra rằng: sử dụng đúng **PHP còn khó hơn các ngôn ngữ khác**.

Trong loạt bài này sẽ lần lượt nêu ra 21 lỗi, từ những sai lầm giáo khoa (làm script trở nên chậm và khó quản lí) đến những sai lầm chết người - có thể xem là nguồn gốc của những sai lầm sơ đẳng.

PHẦN 1: 7 LỖI GIÁO KHOA

21. Sử dụng printf() không thích hợp

Hàm printf() dùng để in dữ liệu có định dạng

Nó có thể được dùng, thí dụ, khi bạn một in một số kiểu double với 2 số lẻ, hoặc trong bất kì tình huống nào bạn muốn thay đổi định dạng trước khi in.

Thí dụ dưới đây minh hoạ cách dùng đúng của printf(): định dạng số Pi với độ chính xác theo ý muốn

Mã lệnh (PHP)

```
<?
        /* The three faces of Π  */
        printf ("Pi is: %.2f\n<br>\n", M_PI);
        printf ("Pi is also: %.3f\n<br>\n", M_PI);
        printf ("Pi is also: %.4f\n<br>\n", M_PI);
?>
```

Chú ý. Tôi đã từng gặp những người sợ dùng printf(), thay vào đó lại dùng những hàm định dạng tự viết, dài đến 30- 40 dòng, trong đi một câu printf() có thể làm mọi thứ anh ta mong muốn.

Nhiều lập trình viên dùng sai printf(): in các biến, các giá trị trả về của hàm hoặc thỉnh thoảng, chỉ là dữ liệu thông thường. Thường xảy ra trong hai tình huống:

- Câu lệnh print() thích hợp hơn
- Hiển thị giá trị trả về của một hàm

1. Khi nào print() thích hợp hơn?

Các lập trình viên thường sử dụng printf() trong khi chỉ print() là đủ. Xét thí dụ sau:

```
<?
        $name = 'Nguyen Anh Khoa';
        $nghenghiep = 'Sinh vien';
        $diachi = 'Phong 204.B5 – KTX – DHSP Dong Thap';
        $email = 'khoa_computer2004@yahoo.com';

        printf ("Ten cua tui la: %s\n<br>\n
                 Hien dang la: %s, %s\n<br>\n
                 Lien he voi toi qua Email: %s\n<br>\n",
                 $name, $nghenghiep, $diachi, $email);
?>
```

Hàm **print()** có thể dùng thay cho printf() như sau:

```
print "Ten cua tui la: $name\n<br>\n Hien dang la: $nghenghiep, $diachi\n<br>\n
      Lien he voi toi qua Email: $email\n<br>\n";
```

Khi không cần định dạng sử liệu, dùng print() thay cho printf() có những lợi ích sau:

- **Thi hành nhanh hơn:** *hàm printf() định dạng dữ liệu của bạn trước khi hiển thị, nó sẽ chậm hơn print() hoặc echo()*
- **Mã sáng sủa:** *hãy xem, dùng hàm printf() sẽ làm cho người đọc hơi bị lẫn lộn (tất nhiên trừ khi họ có nền tảng C). Nó đòi hỏi kiến thức về cú pháp printf() (thí dụ, %s thay cho chuỗi còn %d là số) và về kiểu biến*

2. Dùng printf() để xuất dữ liệu trả về từ gọi hàm

Một lỗi thường gặp khác là dùn printf() để xuất dữ liệu trả về từ gọi hàm, thí dụ như hàm đếm dưới đây:

```
<?
      printf("%d occurrences of %s found.",
            count($truyvan), $search_term);
?>
```

Khi xuất giá trị do hàm trả về, toán tử . nên dùng để nối trong print(), như dưới đây:

```
<?
      print count($truyvan) .
            "occurrences of $search_term found.";
?>
```

Dùng toán tử. nhanh hơn việc dùng printf()

20. Áp dụng sai ngữ nghĩa (semantics)

Nhiều lập trình viên sử dụng PHP mà không biết đến những điểm tinh tế của ngôn ngữ này. Một trong những điểm đó là sự khác nhau giữa cú pháp (syntax) và ngữ nghĩa (semantics).

- **Cú pháp:** *những quy tắc định nghĩa một phần tử. Thí dụ: dấu $ để trước dùng định nghĩa biến, dùng dấu () và các tham số định nghĩa một hàm...*
- **Ngữ nghĩa:** *những quy tắc áp dụng trong cú pháp. Thí dụ: một hàm có 2 biến định nghĩa bởi cú pháp của nó, hai biến này có kiểu string - đó là ngữ nghĩa.*

Trong một ngôn ngữ lỏng lẻo như PHP, bạn có nhiều lựa chọn để viết lệnh. Các biến không cần có kiểu xác định...

Thí dụ sau mở tập tin và in từng dòng:

```
<?
   $fp = @fopen ('vanban.txt', 'r')
   or die ('Khong the mo file vanban.txt');
   while ($line = @fgets ("$fp", 1024)) // Loi {
        print $line;
   }

   @fclose ("$fp") // Loi
   or die ('Khong the dong vanban.txt');
?>
```

Thí dụ trên sẽ tạo lỗi:

Warning: Supplied argument is not a valid File-Handle resource in C:\Inetpub\wwwroot\tst.php on line 4.

Đó là do biến $fp đặt trong dấu nháy kép nên được chuyển thành chuỗi. Thế mà hàm fopen() nhận một định danh tài nguyên (resource identifier) trong tham số đầu của nó, chứ không nhận một chuỗi. Để giải quyết vấn đề, bạn chỉ đơn giản bỏ dấu nháy kép đi

```
<?
        $fp = @fopen ('vanban.txt', 'r')
        or die ('Khong the mo file tailieuphp.txt');
        while ($line = @fgets ($fp, 1024))  {
              print $line;
        }

        @fclose ($fp)
        or die (Khong the dong file vanban.txt ');
?>
```

1. Có thể tránh việc áp dụng sai ngữ nghĩa?

Our example above generated an error statement. But PHP enables you to customize your scripts to fit a unique scenario or output requirement. So, it is at least theoretically possible to "get away" with misapplying a semantic. Tôi không hiểu, nhưng dịch thế này được không?

Thí dụ trên của chúng ta tạo ra một thông báo lỗi. Nhưng PHP cho phép bạn tuỳ biến các script để thích hợp với một kịch bản khác thường hoặc với các đòi hỏi của thông tin ra. Do đó, ít nhất trên lí thuyết, bạn có khả năng tránh việc áp dụng sai ngữ nghĩa.

Vậy, bạn cần biết những hậu quả có thể có (possible outcomes) nếu bạn quyết định học về ngữ nghĩa. Áp dụng sai dẫn đến những lỗi khá tinh vi nếu bạn không chú ý.

Nếu bạn muốn tuỳ biến script, bạn cần hiểu những chủ đề chính sau:

- **Kiểu:** *trong PHP, mỗi biến có một kiểu xác định ở một thời điểm xác định, cho dù bạn có thể tự do chuyển đổi kiểu một biến. Nói một cách khác, không có biến nào lại không kèm theo tính chất của kiểu của nó. PHP có 7 kiểu cơ bản: boolean, resource, integer, double, string, array và object.*
- **Tầm vực:** *trong PHP, mỗi biến có một tầm vực riêng. Tầm vực biến quy định biến có thể được truy cập từ đâu, tồn tại trong thời gian nào. Hiểu sai khái niệm cơ bản về "tầm vực" dẫn đến những lỗi sai tinh tế và cả những lỗi lớn.*
- **php.ini:** *khi viết một script chạy ở nhiều môi trường khác nhau, cần biết rằng không phải mọi cấu hình PHP đều như nhau. Do đó, cần thiết những lệnh kiểm tra để đảm bảo script của bạn chạy tốt trong cấu hình PHP của người khác.*

19. Thiếu ghi chú

Theo ý tôi, mã nguồn thiếu ghi chú là căn nguyên của sự lập trình ích kỉ. Nó dẫn tới những hiệu chỉnh sai lầm, hiểu sai ý nghĩa và làm người đọc mệt mỏi. Nói chung, lập trình ghi chú (inline documentation) được mọi người khẳng nhận là điều tốt, nhưng hiếm khi nó tồn tại.

Một vấn đề khác là quá nhiều ghi chú. Dù hiếm gặp, nhưng nó làm cho các đoạn mã bị cắt vụn, gây ra sự khó theo dõi. Dưới đây là một thí dụ:

```
<?
        // Bat dau ma PHP
        $age = 18; // Gan 18 den $age
        $age++; // Tang $age len 1 don vi
        // IN ra doan text thong bao:
        print "You are now 19, which means you have been:";
        print "\n<br>\n<br>\n";
        // Vong lap for in ra tat ca tuoi truoc do
        for ($idx = 0; $idx < $age; $idx++) {
```

```
            // In ra tuoi ca nhan
            print "$idx years old\n<br>\n";
        } // Ket thuc ma PHP
?>
```

1. Bao nhiêu ghi chú thì đủ?

Nhiếu đến mức nào, điều đó tuỳ thuộc ngân sách của bạn, vào chính sách của công ty và vào độ phức tạp của chương trình. Tuy nhiên, cũng có một vài gợi ý cho bạn

- *Luôn có một mô tả ngắn về mục đích của hàm ngay trước định nghĩa của hàm đó*
- *Thêm ghi chú vào những chỗ có thể bị hack, hoặc những chỗ tưởng rằng sai nhưng lại chạy đúng* 😕
- *Nếu một đoạn mã nào đó có thể gây nhầm lẫn, hãy thêm một ít ghi chú về mục đích của đoạn đó. Sau này bạn sẽ thấy được lợi ích của nó*
- *Dùng một kiểu ghi chú nhất quán,* ***/* */*** *hoặc là* ***// (tránh dùng #)***

Dưới đây là một thí dụ về ghi chú tốt:

```
<?
    // Random_Numbers.lib
    // Generate different types of random numbers.

    mt_srand((double)microtime()*1000000);
    // mixed random_element(array elements[, array weights])
    // Extract a random element from elements. Weights is
    // the relative probability that each element will be
    // selected.
        function random_element ($elements, $weights=array()) {
            // There must be exactly the same amount of elements as
            // there are weights for this algorithm to work properly

            if (count ($weights) == count ($elements)) {
                foreach ($elements as $element) {
                        foreach ($weights as $idx) {
                            // Note: we don't use $idx, since we
                            // don't want to override elements.
                            $randomAr[] = $element;
                        }
                }
            } else {
                $randomAr = $elements;
            }
            $random_element = mt_rand (0, count ($randomAr)-1);
            return $randomAr[$random_element];
    }
?>
```

18. Nhiều biến, tốn nhiều thời gian

Có vài người bị ám ảnh bởi biến trung gian. Tôi không thể hiểu nổi tại sao ai đó có thể viết như thế này:

```
<?
    $tmp = date ("F d, h:i a"); /* ie January 3, 2:30 pm */
    print $tmp;
?>
```

Tại sao phải dùng biến trung gian? Nó không cần thiết

```
<?
    print date ("F d, h:i a");
?>
```

Rủi thay, có vẻ như rất nhiều người khó bỏ được thói quen xấu này.

Biến tạm làm chậm thời gian thi hành chương tình của bạn. Tốt hơn là nên bỏ qua đó và gộp các lời gọi hàm với nhau. Những người dùng biến tạm thường làm chương của họ chạy chậm đến 25%.

Một lí do khác để tránh có quá nhiều biến tạm là vì trông nó không được đẹp mắt. Trong hai thí dụ trên, thí dụ nào súc tích hơn? Thí dụ nào làm con mắt dễ chịu hơn? Dùng quá nhiều biến tạm có thể dẫn đến mã chương trình khó đọc và không súc tích.

1. Lợi điểm của dùng biến tạm

Các biến tạm có lợi trong việc thay thế các hàm hay biểu thức dài lê thê. Nó có vai trò như bí danh giả. Điều này đặc biệt đúng khi bạn dùng một hàm hay biểu thức nhiều lần.

Xem xét thí dụ đây, nó không dùng nhiều biến hơn mức tối thiểu

```
// string reverse_characters(string str)
// Reverse all of the characters in a string.
function reverse_characters ($str) {
     return implode ("", array_reverse (preg_split("//", $str)));
}
```

Nội dung trong hàm implode() dài và do đó khó đọc. Dùng một hoặc nhiều biến tạm có thể giúp chúng ta:

```
// string reverse_characters(string str)
// Reverse all of the characters in a string.
function reverse_characters ($str) {
       $characters = preg_split ("//", $str);
       $characters = array_reverse ($characters);
       return implode ("", $characters);
}
```

2. Các luật chung của ngón tay cái

Khi quyết định có dùng biến tạm hoặc không, bạn nên suy nghĩ về 2 câu hỏi:

- Bạn có dùng biến đó ít nhất hai lần?
- Tính đọc được của mã có tăng đáng kể không?

Nếu ít nhất một câu trả lời là có, thì nên dùng biến tạm. Còn không, vứt nó đi và tổ hợp các hàm lại (nếu cần).

17. Viết lại các hàm có sẵn

Một số nơi phổ biến mã nguồn các script PHP chủ trương đổi tên các hàm sẵn có để tạo sự dễ dàng cho các lập trình viên chuyển từ VB sang. Thí dụ:

```
<?
      function len ($str)  {
```

```
            return strlen ($str);
        }
?>
```

Lại có một số người cố gắng viết lại các hàm PHP thông dụng thay vì đi học về hàm đó trong các tài liệu PHP cung cấp.

Có ít nhất 2 lí do để không nên làm điều này. Thứ nhất, và trên nhất, nó làm cho những người đọc (và sửa) chương trình của bạn khó hiểu và cảm thấy có quá nhiều hàm dư thừa. Họ tự hỏi tại sao bạn lại đi định nghĩa hàm theo kiểu đó, thay vì sử dụng các hàm định nghĩa sẵn bởi PHP.

Thứ hai, định nghĩa hàm như vậy cũng sẽ làm chậm chương trình của bạn (một cách không cần thiết). Không chỉ phải xử lí nhiều mã hơn, mà mỗi lần gọi hàm do bạn định nghĩa, bạn đã tốn thời gian cho chính hàm đó, trước khi hàm nguyên thuỷ được gọi.

1. Tránh viết lại các hàm có sẵn

Hãy đương đầu với nó. Đôi khi thật là khó để tránh chuyện này. Trước tiên, một lập trình viên không thể theo kịp các hàm của PHP ngay được. Và ai có thời gian mà tra cứu. Tại sao không viết lại cho khoẻ?

Cách làm của tôi là luôn có sẵn một tài liệu chỉ dẫn PHP (*PHP manual*) mỗi khi viết chương trình (tác giả bài này dùng một bản PDF có tạo chỉ mục, riêng tôi, người dịch, thì dùng một tài liệu CHM đầy đủ thông tin và có cả góp ý của người sử dụng mà bạn có thể lấy ở ~~http://www.php.net/docs.php)~~. Sau đó, mỗi khi định viết một hàm *mở rộng* cho PHP, tôi đọc lướt qua tài liệu để xem hàm đó có chưa.

Tuy nhiên, cần chú ý là, do bản chất mã nguồn mở của PHP, bạn có thể tìm được các hàm do người dùng định nghĩa trước khi nó được thêm vào PHP (thí dụ như hàm tìm phần tử khác nhau giữa hai mảng). Điều này không có nghĩa là bạn phải hiệu chỉnh lại mã (This doesn't necessarily mean that you should have to correct the code. - don't understand)

16. Không tách biệt phần server và client

Vài lập trình viên cố kết nối cả chương trình với nhau, nghĩa là ghép chung mã HTML (client-side - phần khách) với mã PHP (server-side - phần chủ) vào trong một tập tin lớn.

Mặc dù điều này tốt cho các site nhỏ, nhưng nó có thể trở thành vấn đề lớn khi các site đó trở nên lớn hơn và được bổ sung thêm tính năng. Lập trình theo cách này làm nảy sinh vấn đề khó bảo trì và các tập tin trở nên cồng kềnh.

1. Hàm API

Khi muốn tách biệt phần khách - chủ, bạn có vài lựa chọn. Một cách là viết những hàm hiển thị nội dung linh động và đặt chúng đúng chỗ trong trang web.

Thí dụ dưới đây minh hoạ điều này:

index.php - phần khách

HTML

```
<?php include_once ("site.lib"); ?>
<html>
<head>
<title> <?php print_header (); ?> </title>
</head>
<body>
    <h1> <?php print_header (); ?> </h1>
        <table border="0" cellpadding="0" cellspacing="0">
            <tr>
```

```
            <td width="25%">
            <?php print_links (); ?>
            </td>
            <td>
            <?php print_body (); ?>
            </td>
            </tr>
        </table>
</body>
</html>
```

site.lib - phần chủ

```
<?php
    $dbh = mysql_connect ("localhost", "khoa", "pass")
    or die (sprintf ("Khong the ket noi den MySQL [%s]: %s",
    mysql_errno(), mysql_error()));
    @mysql_select_db ("MainSite")
    or die (sprintf ("Khong the chon CSDL![%s]: %s", mysql_errno(),
    mysql_error()));

    $sth = @mysql_query ("SELECT * FROM site", $dbh)
    or die (sprintf ("Khong the thuc hien truy van [%s]: %s",
    mysql_errno(), mysql_error ()));

    $site_info = mysql_fetch_object($sth);

    function print_header () {
       global $site_info;
       print $site_info->header;
    }

    function print_body () {
       global $site_info;
       print nl2br ($site_info->body);
    }

    function print_links () {
       global $site_info;

       $links = explode ("\n", $site_info->links);
       $names = explode ("\n", $site_info->link_names);

       for ($i = 0; $i < count ($links); $i++) {
          print "\t\t\t <a xhref=\"$links[$i]\">$names[$i]</a>\n<br>\n";
       }
    }
?>
```

Như bạn thấy trong thí dụ trên, tách biệt khách chủ làm tăng tính dễ đọc trong chương trình của bạn. Một lợi ích khác là một khi bạn đã có các hàm API hiển thị nội dung, bạn có thể để cho thiết kế viên tham gia thay đổi bố cục mà không cần sửa mã chương trình.

1.1. Lợi ích của hàm API

- Tương đối sáng sủa
- Nhanh, hầu như không lãng phí thời gian (overhead)

1.2. Bất lợi

- Không sáng sủa và dễ dàng bằng hệ thống mẫu (template system)
- Cần một ít kiến thức PHP để sửa mẫu

2. Hệ thống khuôn mẫu

Một cách khác để tách biệt khách chủ là dùng hệ thống khuôn mẫu. Nghĩa là, có một số đánh dấu nội dung sau đó dùng chương trình phân tích, thay thế các đánh dấu đó bằng thông tin cần thiết.

Thí dụ, bạn có thể tạo một tập tin như thế này:

HTML

```
<html>
<head>
<title>%%PAGE_TITLE%%</title>
</head>
<body %%BODY_PROPERTIES%%>
    <h1>%%PAGE_TITLE%%</h1>
    <table border="0" cellpadding="0" cellspacing="0">
        <tr>
        <td width="25%">%%PAGE_LINKS%%</td>
        <td>%%PAGE_CONTENT%%</td>
        </tr>
    </table>
</body>
</html>
```

Sau đó có thể viết chương trình phân tách tập tin, thay thế các thông tin trong dấu cách %% bằng các thông tin thích hợp.

Ghi chú: một lớp hỗ trợ hệ thống khuôn mẫu khá tốt là lớp FastTemplate, có ở www.thewebmasters.net

2.1. Ưu điểm của hệ thống khuôn mẫu

- Rất trong sáng
- Không cần kiến thức PHP để sửa khuôn mẫu

2.2. Nhược điểm

- Chậm hơn, bạn cần phân tách tập tin khuôn mẫu, sau đó xuất ra
- Việc hiện thực phức tạp hơn

15. Dùng các cấu trúc lỗi thời

Có nhiều người cứ dùng mãi các mã và thư viện lỗi thời. Thí dụ như họ đã viết một hàm dùng ở PHP 2, và vẫn còn dùng nó ở PHP 4, mặc dù một hàm có cùng mục đích như thế đã được thêm vào ở PHP 3 

Dùng các cấu trúc lỗi thời có thể làm chậm chương trình của bạn, cũng như làm cho nó trở nên khó hiểu. Người đọc các chương trình của bạn có thể không quen với các hàm lỗi thời của PHP. Tuy nhiên, khi phát hiện một đoạn mã lạc hậu, bạn đừng nghĩ rằng cần phải thay thế nó. Chỉ cần chắc chắn rằng bạn sẽ không dùng nó cho các chương trình viết trong tương lai.

Một thí dụ về cấu trúc lỗi thời, mà nhiều người có vẻ cố nắm lấy, là cú pháp **beginControlStructure .. endControlStructure**;

```
<?
```

```
        // Bad/Outdated Practice
        while (1):
        print "5";
        if ($idx++ == 5):
        break;
        endif;
        endwhile;

        // Better Practice
        // (the code could be optimized though)
        while (1) {
            print "5";
            if ($idx++ == 5) {
                break;
            }
        }
    ?>
```

Đây là một thói quen xấu vì

- *Nó không được dùng rộng rãi, cho nên nhiều người học sẽ bị lẫn lộn giữa hai cú pháp*
- *Nó không tương thích với ngôn ngữ khác, nghĩa là nó trở nên khó đọc đối với những người trong giai đoạn quá độ (mới chuyển từ một ngôn ngữ nào đó sang PHP)*
- *Quan trọng nhất, là một ngày nào đó tính năng này sẽ bị xoá xổ, bắt buộc bạn phải viết lại toàn bộ mã có dùng nó. Dấu ngoặc nhọn luôn luôn là một phần của ngôn ngữ PHP.*

Ở trên chỉ là một thí dụ về cấu trúc lỗi thời. Nó còn nhiều nữa. Như một quy tắc, bạn nên theo những các viết trong tài liệu PHP. Hầu hết nó được cập nhật mới. Nó cũng dùng các hàm mới nhất của PHP trong thí dụ của mình. Nên thường xuyên kiểm tra tài liệu khi bạn có ý muốn mở rộng tính năng nào đó của PHP. Theo cách này, bạn sẽ không phải viết lại các hàm có sẵn.

Tổng kết

Trong bài này bạn đã đi qua 7 trên tổng số 21 lỗi mà lập trình viên PHP mắc phải. Những lỗi giáo khoa này bao gồm:

- **Sử dụng sai hàm printf()**
- **Áp dụng sai ngữ nghĩa**
- **Thiếu tài liệu trong mã nguồn**
- **Dùng quá nhiều biến tạm**
- **Viết lại các hàm có sẵn**
- **Không tách biệt phần khách/chủ**
- **Dùng các cấu trúc lỗi thời**

PHẦN 2 – LỖI NGHIÊM TRỌNG

14. Không tuân thủ các quy ước đặt tên

Một trong những lỗi nghiêm trọng mà người lập trình có thể phạm phải là định nghĩa một quy ước đặt tên tồi. Tôi đã tiếp quản nhiều dự án mà trong đó tôi phải bỏ ra rất nhiều thời giờ chỉ để hiểu chương trình, do lập trình viên đặt tên các biến là $fred và $barney thay cho $email và $name. Tôi đang đề cập đến một dự án mà người lập trình cũ đã quyết định đưa vào *toàn bộ* chương trình một kiểu đặt tên kì lạ (a Flinstones naming theme), không phải tôi đùa đâu.

Cách bạn đặt tên biến và hàm là trung tâm của việc xây dựng một chương trình dễ đọc. Có nhiều lập trình viên phạm lỗi khi đặt tên biến và hàm mà nó:

- quá dài hoặc quá ngắn
- không liên quan đến ngữ cảnh
- không để ý đến cách-viết-phân-biệt (case sensitivity)
- ngăn cản khả năng dễ đọc (đặc biệt là các hàm)

1. Đặt tên biến

1.1. Cách viết phân biệt

Trong PHP, tên biến có cách viết phân biệt, nghĩa là $user và $User là hoàn toàn khác nhau. Vài người dùng lợi dụng điểm này để đặt các biên cùng tên nhưng khác cách viết. Đây là một thói quen tồi tệ. Cách viết không bao giờ nên dùng để phân biệt các biến khác nhau. Mỗi tên biến, trong cùng tầm vực (scope), nên có là tuyệt đối duy nhất.

1.2. Tên quá ngắn

Nhiều người sử dụng những chữ viết tắt đầu (cryptic acronym) bí ẩn cho các biến của họ, để rồi sau này hối tiếc vì quên mất họ đã muốn ám chỉ điều gì khi đó. Tên biến nên mô tả nội dung nó (sẽ) chứa, dùng nguyên từ hoặc những chữ viết tắt có thể hiểu được.

1.3. Tên quá dài

Ở khía cạnh khác, vài người lại sử dụng tên biến quá dài. Nói chung, tên biến không nên dài quá hai từ. Hai từ có thể được tách biệt bằng dấu phân cách "_" hoặc là viết hoa chữ đầu của từ thứ hai.

1.4. Thói quen tốt

Dưới đây là những thí dụ tốt về tên biến

```
$username = 'phanthanhkieu';
$password = 'bimat';
$teachers = array ('Sadlon',
                   'Lane',
                   'Patterson',
                   'Perry',
                   'Sandler',
                   'Mendick',
                   'Zung');
foreach ($teachers as $teacher);
```

1.5. Thói quen xấu

Dưới đây là những thí dụ (phóng đại) về những tên biến tồi

```
$username_cua_csdl= 'SINHVIEN';
$guMbi = 'bimat'; // for the $password

$tentruocdo_cua_giaovien = array ('Sadlon',
                                  'Lane',
                                  'Patterson',
                                  'Perry',
                                  'Sandler',
                                  'Mendick',
                                  'Zung');
foreach ($tentruocdo_cua_giaovien as $TeaChER);
```

2. Đặt tên hàm

Mọi khái niệm áp dụng cho tên biến cũng áp dụng cho đặt tên hàm. Tuy nhiên, ngữ pháp đóng vai trò đặc biệt trong các hàm.

Các hàm PHP, định nghĩa sẵn hoặc do người dùng định nghĩa, là không-phân-biệt-cách-viết (not case sensitive)

2.1. Dùng động từ

Hàm của PHP tương đương với một động từ khi nói. Tên hàm, do đó, nên được hướng hành động (action oriented). Nó cũng nên được dùng ở thì hiện tại.

Thí dụ, bạn có một hàm tạo một số ngẫu nhiên với phân bố Gausse (a gaussian random number), bạn nên đặt tên nó là generate_gaussian_rand().

Chú ý các sử dụng động từ hành động trong tên hàm. Nó sẽ đặt hàm vào ngữ cảnh thích hợp

```
<?php
     list ($num1, $num2) = generate_gaussian_rand();
     list ($num3, $num4) = generate_gaussian_rand();
?>
```

Để so sánh, hãy xem thí dụ:

```
<?php
     list ($num1, $num2) = gaussian_rand_generator();
     list ($num1, $num2) = gaussian_rand_generator();
?>
```

Bạn có thấy sự khác biệt? Thí dụ thứ hai sử dụng danh từ, mặc dù vẫn chuyển tải được mục tiêu của hàm, nhưng nó ngăn người ta đọc một cách trôi chảy.

Hãy sử dụng động từ!

13. Không suy nghĩ thấu đáo: CSDL & SQL

Số cách người ta truy cập cơ sở dữ liệu (CSDL - database) và lấy kết quả nhiều đến mức thực sự ngạc nhiên. Những thí dụ tôi đã gặp bao gồm những tổ hợp lệnh if và vòng lặp do.. while, các câu gọi nhiều lần, và các hàm sql_result() trong vòng for.

Những người này có *nghĩ* họ đang làm gì không?

Việc viết các mã trật-hoặc-trúng (hit-or-miss code) chứng minh sự thiếu tập trung. Những cá nhân đó xác định nỗ lực của họ dùng để hoàn thành công việc hơn là để hoàn thành *đúng* công việc, kết quả là làm cho các ông chủ quăng thời gian và tiền bạc ra đường.

Sự lấy mẫu không chính xác là một thí dụ hay về vấn đề này. Vài người viết lệnh không dành thời gian để nghĩ thấu đáo. Đúng là không chỉ có duy nhất một cách "đúng" để lấy mẫu dữ liệu, nhưng nó có rất nhiều cách không đúng.

Phần này bao gồm các chủ đề:

- Áp dụng sai các hàm về CSDL
- Dùng sai SQL: không lấy những thứ bạn cần
- Dùng PHP để sắp xếp kết quả

1. Dùng sai các hàm CSDL

Một đoạn mã PHP đã dùng cú pháp sau để lấy kết quả từ CSDL (presented below using a *generalized* set of SQL functions):

```
if (!($mautin = sql_fetch_row ($truyvan))) {
    print "Mot loi xay ra: Khong tim thay mau tin nao!";
    exit;
}
do {
    print "$mautin[0]: $mautin[1]\n<br>\n";
} while ($mautin = sql_fetch_row ($truyvan));
```

Chú ý: Ở trên, và các thí dụ sau nữa, $truyvan diễn tả handle hoặc pointer đến một tập kết quả truy vấn.. Nói cách khác, một truy vấn đã được gửi và một tập kết quả đã được trả về. Các thí dụ sẽ nói về vấn đề thao với với kết quả trả về.

Có một vài vấn đề với đoạn mã trên:

- Nó kiểm tra các trường hợp "không tìm thấy" ("no incidents" case) bằng cách lấy một dòng
- Nó không lưu kết quả vào mảng liên kết (associative array)

1.1. Kiểm tra trường hợp không tìm thấy: cách làm sai

Bằng cách dùng sql_fetch_row(), PHP chủ trương một cách tiếp cận hàm ẩn cho việc xác định có kết quả tìm thấy hay không. Một cách khác trực tiếp và tường minh là đếm số dòng của kết quả bằng sql_num_rows() như dưới đây:

```
<?php
    if (sql_num_rows ($truyvan) <= 0) {
        print "Mot loi xay ra: Khong tim thay mau tin nao!";
        exit;
    }
    while ($mautin = sql_fetch_row ($truyvan)){
        print "$mautin[0]: $mautin[1]\n<br>\n";
    }
?>
```

1.2. Từ bỏ vòng lặp Do..While

Trước hết và trên hết, vòng lặp thô tục do..while không bao giờ cần nữa vì khi dùng sql_num_row(), chúng ta không phải lấy dòng đầu tiên của kết quả khi muốn kiểm tra kết quả trống.

Thí dụ cũ đã diễn tả một thí dụ mà trong đó, nếu kết quả không rỗng, dòng đầu tiên đã được lấy bằng hàm sql_fetch_row() trong câu lệnh if. Cấu trúc do..while cần thiết trong trường hợp này vì khi đó, bộ đếm của CSDL đã tăng lên và chuyển sang dòng kế tiếp. Do đó, bạn phải *xử lí* (lệnh do)dòng đầu tiên vì nó đã được lấy. Các lệnh tiếp theo lấy các dòng kế, và cứ thế.

Tại sao vòng **do..while** bị coi như thô tục, xấu xa (nasty)?

- *Trong thí dụ trên, chỉ có 1 lệnh in kết quả trong vòng lặp. Hãy tưởng tượng, nếu có 10 lệnh, người đọc mã phải tìm điều kiện while* sau *các câu lệnh đó. Một công việc phiền toái.*
- *Điều kiện While bắt đầu cấu trúc thường xuyên hơn là kết thúc. Một nhà nghiên cứu sẽ phải cẩn thận hơn để không nhầm lẫn điều kiện while* cuối *với điều kiện while* đầu*.*

1.3. Giữ mọi thứ gọn gàng và đơn giản

Với trường hợp kết quả rỗng, sql_num_rows()đưa đến ngay kết quả, trong khi sql_fetch_row() thì không

- **sql_fetch_row()** nói rằng "*Tôi* tìm thấy *0 dòng trong tập kết quả. Điều này nghĩa là* có *0 kết quả*" (I found no rows in the result set. This must mean that there are none.)
- **sql_num_rows()** nói rằng "*Số dòng trong kết quả* là *0*" (The number of rows in the result set is 0).

Nhưng điều đó thực sự tạo nên sự khác biệt nào?

Xét cùng một sự so sánh, nhưng bây giờ là trong ngữ cảnh của điều kiện if và của biểu thức, trong đoạn lệnh giả (Pseudo-code):

*** if(!($mautin = sql_fetch_row($truyvan))) { Print Error }:**

- lấy 1 dòng trong tập kết quả
- nếu kết quả rỗng, gán cho $mautin giá trị zero (0); 0 có giá trị logic là False, do đó !(0) = True; in thông báo lỗi.
- nếu không rỗng, lấy dòng đầu tiên và gán nó vào $mautin; $mautin không phải là zero và có giá trị là True. Do đó !(True) = False, và tiếp tục với cấu trúc do..while.

*** if((sql_num_rows($truyvan)<= 0) { Print Error }:**

- đếm số dòng trong tập kết quả.
- Nếu nhỏ hơn hay bằng 0, in thông báo lỗi.
- Nếu không, tiếp tục.

Biểu thức nào dể hiểu hơn? Rõ ràng là cách đếm sẽ trực tiếp và gọn gàng hơn.

Sự khác biệt thực tế là gì? Với một lệnh if đơn giản, chúng ta không thu được lợi nhều.

Tuy nhiên, với hơn 10 000 dòng lệnh, hãy dành thời gian *nghĩ* đến cách rõ ràng nhất, nó sẽ tiết kiệm cho người phân tích chương trình nhiều giờ suy nghĩa. Lợi ích khác có thể kể đến là chương trình của bạn sẽ nhanh hơn và dễ phát triển hơn.

1.4. Khi mà DBMS của bạn không hỗ trợ sql_num_row()

Vài DBMS có thể không hỗ trợ hàm sql_num_row(). Tôi xin chia sẻ với bạn nếu DBMS của bạn là một trong số đó. Bạn sẽ phải tìm trong kết quả rỗng bằng cách lấy dòng. Tuy nhiên, trong trường hợp này, nó nên dùng một biến boolean như sau:

```
<?php
     $timthay= false;
     while ($mautin = sql_fetch_array($truyvan)){
         $timthay= true;
     }

     if (!$timthay){
         print "Loi!";
     }
?>
```

1.5. Lấy kết quả: hãy chọn cách có ích

Vấn đề thứ hai trong đoạn mã này là nó dùng sql_fetch_row() để lấy tập kết quả. Hàm sql_fetch_row() *chỉ* trả về mảng đánh chỉ số, trong khi đó sql_fetch_array() trả về mảng đánh chỉ số *và* mảng dùng chuỗi.

```
$mautin = sql_fetch_array ($truyvan);
print $mautin[1];      // Cot thu 2
print $mautin[name];   // Ten cot
```

Chú ý: *Có nhiều quy ước khác nhau về việc dùng dấu nháy khi thêm một đối số kiểu chuỗi. Trong thí dụ về tên cột ở trên, và suốt bài viết này, nó sẽ được bỏ.*

Từ quan điểm của nhà phát triển, hàm nào có lợi hơn? Mảng dùng chuỗi giúp cho người đọc hiểu được bạn đang lấy cái gì chỉ thông qua việc đọc mã, như thí dụ đúng dưới đây:

```
<?php
     if (sql_num_rows($truyvan)<= 0) {
         print "Mot loi da xay ra: Khong co mau tin nao!";
         exit;
     }

     while ($mautin = sql_fetch_array ($truyvan)) {
         print "$mautin[name]: $mautin[phone_number]\n<br>\n";
     }
?>
```

1.6. Khi nào sql_fetch_row($truyvan) nên được dùng

Tôi không thực sự là fan của the sql_fetch_row(). Tuy nhiên, có một tình huống mà dùng nó không giảm khả năng dễ đọc: khi người dùng định nghĩa câu truy vấn.

Các thí dụ cho đến lúc này đều đề cập đến những câu truy vấn được biết trước. Đôi khi bạn để cho người dùng tự định nghĩa câu truy vấn. Trường hợp này bạn sẽ không biết các cột trong kết quả.

Do đó, dùng hàm sql_fetch_row() kèm với count() sẽ xử lí hiệu quả các cột trong một hàng:

```
<?php
    for ($i = 0; $i < count($mautin); $i++){
        print "Column". ($i+1). $mautin[$i]. "\n<BR>\n";
    }
?>
```

2. Dùng sai SQL: không lấy những gì bạn cần

Như là vấn đề của thực hành, đơn giản là sẽ sai lầm khi dùng PHP xử lí mọi dòng của CSDL. Tôi đã bắt gặp người ta dùng PHP để chạy một chương trình tìm kiếm đơn giản trên 2MB dữ liệu và tự hỏi tại sao cái ngôn ngữ này chạy lâu thế. Lấy 2MB dữ liệu từ CSDL có thể làm bạn chờ mãi mãi.

Ngôn ngữ truy vấn chuẩn (**Standard Query Language - SQL**) được thiết kế đặc biệt để truy vấn và lấy dữ liệu từ các bảng của bạn. Ý tưởng là dùng nó để lọc dữ liệu không cần thiết, để lại các thông tin liên quan cho PHP xử lí.

Nếu bạn lấy nhiều dữ liệu hơn cần thiết, đó là dấu hiệu chắc chắn rằng mã SQL đang dùng chưa được tối ưu hoá.

2.1. Mệnh đề WHERE

Một thí dụ kinh điển về sự hiểu quả của SQL liên quan đến mệnh đề where.

Đoạn mã sau sẽ lấy các kết quả và in ra tên và mã sinh viên của sinh viên có MASV='511203008':

```
<?php
    include("includes/taptinketnoi.inc");

    $strsql = "SELECT MASV, HOTEN FROM SINHVIEN";
    $truyvan = @sql_query ($strsql, $ketnoi);

    if (!$truyvan) {
        die (sprintf ("LOI: [%d]: %s",
        sql_errno (), sql_error ()));
    }

    if (@sql_num_rows ($truyvan) <= 0) {
        die ("Khong ket qua tim duoc tu CSDL!");
    }

    while ($mautin = @sql_fetch_array ($truyvan)){
        if ($mautin[MASV] == "511203008") {
            print "MASV: $mautin[MASV]\n<br>\n";
            print "Ho ten: $mautin[HOTEN]\n<br>\n";
            break;
        }
    }
?>
```

Đoạn mã trên chưa được tối ưu: chúng ta đang dùng PHP để tìm kiếm trong toàn bộ CSDL! Nếu như điều này không quan trọng đối với các CSDL nhỏ, khi kích thước CSDL tăng lên bạn sẽ cảm thấy một cú đấm nặng nề về hiệu năng.

Lời giải rất đơn giản: sửa câu SQL để chứa mệnh đề WHERE:

```
$strsql = "SELECT * FROM SINHVIEN";
$strsql .= " WHERE MASV='511203008'";
```

Mệnh đề WHERE cho phép bạn tìm kiếm chọn lọc hơn. Giới hạn chọn lọc của mệnh đề where chính là một hàm với đối số của nó. Trong thí dụ trên đối số là " MASV='511203008'".

Bây giờ chúng ta đã chọn được dữ liệu cần thiết, bạn chỉ việc dùng PHP để in ra sau đó:

```
if (@sql_num_rows ($truyvan) != 1) {
    die ("Khong dung so mau tin nhan duoc tu CSDL!");
}

$mautin = @sql_fetch_array ($truyvan);
print "Ho va ten: $mautin[HOTEN]\n<br>\n";
print "MASV: $mautin[MASV]\n<br>\n";
```

3. Dùng PHP sắp xếp kết quả

Nhiều người lấy dữ liệu ở tình trạng không có thứ tự, nhưng rồi đoạn mã PHP tiếp theo lại sắp thứ tự chúng. Nên chớ rằng sắp xếp bằng SQL nhanh hơn PHP.

Dùng cú pháp ORDER BY của SQL để sắp xếp thay vì hàm ksort() của PHP.

Thí dụ dưới đây dùng ksort() để sắp xếp theo tên:

```
$strsql = "SELECT name, email, phone FROM some_table ";
$strsql .= "WHERE name IS LIKE '%baggins'";

$truyvan = @sql_db_query ($strsql, "samp_db", $ketnoi);

if (!$truyvan) {
    die (sprintf ("Error [%d]: %s",
    sql_errno (),sql_error ()));
}
while ($mautin = @sql_fetch_array $truyvan)){
   $matches[$mautin[name]] = array ($mautin[email],
   $mautin[phone]);
}
ksort ($matches);
```

Nhưng tại sao không sắp xếp dữ liệu *ngay vào lúc* nó được định nghĩa? Nó giúp chúng ta đỡ phải duyệt qua tập kết quả lần thứ hai.

Do vậy, bỏ hàm ksort() ra khỏi chương trình trên và thay đoạn mã SQL bằng đoạn dưới đây, có dùng dùng cú pháp ORDER BY:

```
$strsql = "SELECT name, email, phone FROM some_table ";
$strsql .= "WHERE name IS LIKE '%baggins' ORDER BY name";
```

12. Thiếu sự kiểm lỗi

Tôi đã thấy nhiều chương trình thiếu một lượng kiểm tra lỗi đầy đủ. Nguyên nhân phần lớn là do lập trình viên không dành thời gian để lên một kế hoạch thích hợp cho chương trình của mình, và xác định những vị trí có thể dẫn đến lỗi. Kiểm tra lỗi không nên thực hiện *sau khi* viết chương trình. Sự thiếu sót trong tầm nhìn trước có thể dẫn đến những lỗi nghiêm trọng, không những gây ra kết quả sai mà thậm chí còn làm hỏng hệ thống (even cause your system to crash)!

1. Mong đợi điều tệ nhất

Mọi chương trình đều có khả năng hư hỏng trong những tình huống *sai*. Để giảm thiểu những rủi ro như thế, bạn cần lên kế hoạch để:

- Kiểm tra kết quả lời gọi hàm
- Kiểm tra kết quả lời gọi hệ thống
- Đặt mức error_reporting là E_ALL trong tập tin php.ini

1.1. Kiểm tra kết quả lời gọi hàm

Mỗi khi bạn gọi một hàm làm thay đổi nhiều dữ liệu, luôn kiểm tra để đảm bảo rằng kết quả trả về trong phạm vi giá trị được chấp nhận (a range of allowable values).

Trong thí dụ dưới đây, một lỗi illegal division by zero sinh ra trong lần lặp thứ 6 của vòng for ($i được tăng lên 1 trong khi $j bị giảm đi 1). Vào lần thứ 6, khi đó $i = $j = 0.

```
<?php
    mt_srand((double)microtime() * 10000000);
    function do_math ($a, $b) {
        return (($a - $b) * 2) / mt_rand();
    }
    for ($i = 5, $j = -5; $i > -5; $i--, $j++){
        print $j / do_math ($i, $j) . "\n";
    }
?>
```

12. Thiếu sự kiểm lỗi

Tôi đã thấy nhiều chương trình thiếu một lượng kiểm tra lỗi đầy đủ. Nguyên nhân phần lớn là do lập trình viên không dành thời gian để lên một kế hoạch thích hợp cho chương trình của mình, và xác định những vị trí có thể dẫn đến lỗi. Kiểm tra lỗi không nên thực hiện *sau khi* viết chương trình. Sự thiếu sót trong tầm nhìn trước có thể dẫn đến những lỗi nghiêm trọng, không những gây ra kết quả sai mà thậm chí còn làm hỏng hệ thống (even cause your system to crash)!

1. Mong đợi điều tệ nhất

Mọi chương trình đều có khả năng hư hỏng trong những tình huống *sai*. Để giảm thiểu những rủi ro như thế, bạn cần lên kế hoạch để:

- Kiểm tra kết quả lời gọi hàm
- Kiểm tra kết quả lời gọi hệ thống
- Đặt mức error_reporting là E_ALL trong tập tin php.ini

1.1. Kiểm tra kết quả lời gọi hàm

Mỗi khi bạn gọi một hàm làm thay đổi nhiều dữ liệu, luôn kiểm tra để đảm bảo rằng kết quả trả về trong phạm vi giá trị được chấp nhận (a range of allowable values).

Trong thí dụ dưới đây, một lỗi illegal division by zero sinh ra trong lần lặp thứ 6 của vòng for ($i được tăng lên 1 trong khi $j bị giảm đi 1). Vào lần thứ 6, khi đó $i = $j = 0.

```
<?php
    mt_srand((double)microtime() * 10000000);
    function do_math ($a, $b) {
        return (($a - $b) * 2) / mt_rand();
    }
    for ($i = 5, $j = -5; $i > -5; $i--, $j++){
        print $j / do_math ($i, $j) . "\n";
    }
?>
```

1.2. Kiểm tra kết quả lời gọi hệ thống

Luôn đảm bảo rằng, khi bạn làm việc với các tiến trình hoặc tập tin ngoài PHP, mọi thứ đều vận hành đúng.

Một thí dụ tuyệt vời là việc kiểm tra đầu ra của một lời gọi hệ thống khi dùng hàm sql_connect(). Xác nhận đầu ra để kiểm tra liên kết đến CSDL là đúng. Làm sai điều này có thể dẫn đến các truy vấn hỏng và mất dữ liệu trong khi thậm chí bạn không biết.

```
$ketnoi = @sql_connect ($host, $user, $pass);
if (!$ketnoi) {
    die (sprintf ("Error [%d]: %s", sql_errno (), sql_error ()));
}
```

1.3. Đặt mức error_reporting là E_ALL trong tập tin php.ini

Hãy đảm bảo bạn cấu hình với mức độ báo lỗi cao nhất có thể. Nếu bạn không đặt nó ở mức cao nhất, ít nhất là trong quá trình tìm lỗi (debugging), bạn có thể bỏ qua những lỗi như là biểu thức chính quy (regular expressions) không hợp lệ và các giá trị không chính xác.

Xem lại lần nữa thí dụ tôi đã đưa trong phần Kiểm tra kết quả lời gọi hàm, ở dưới đây. Giả sử bạn đặt error reporting ở mức thấp,E_ERROR.

Chú ý rằng kết quả in ra khi chương trình thi hành hàm do_math: không có thông báo illegal division by zero đã từng hiện ra lần trước, phần $i=$j=0 đơn thuần không hiện kết quả.

```
<?php
     error_reporting (E_ERROR);
     mt_srand ((double)microtime() * 1000000);
     function do_math ($a, $b) {
          return (($a - $b) * 2) / mt_rand();
     }
     for ($i = 5, $j = -5; $i > -5; $i--, $j++) {
          print $j / do_math ($i, $j) . "\n";
     }
?>
```

Kết quả hiện ra:

```
-5148.25
-5271
-323.75
-4931
-7713.5
?
-4702.5
-488.5
-928.5
-1394.75
```

2. Bộ quản lí lỗi tuỳ chỉnh

PHP thường hiển thị các lỗi thực thi (execution errors) ra trình duyệt, ngăn bạn xoá (suppress) hoặc bắt (capture) nó. Tuy nhiên, với PHP4 bạn đã có thể bắt lỗi bằng hàm set_error_handler().

Hàm set_error_handler() có thể được dùng để ghi lại các lỗi xảy ra với chương trình của bạn. Thay vì làm phiền người dùng với các thông báo lỗi, bạn có thể ghi lại cho riêng bạn, bằng cách đặt một hàm quản lí lỗi tuỳ chỉnh (a custom error handling function).

Trong thí dụ dưới, set_error_handler() được dùng để chỉ định hàm error_handler() là bộ quản lí lỗi mặc định.

Khi một lỗi xảy ra, error_handler() được gọi và hàm PHP error_log() được dùng để ghi lỗi vào tập tin error_file.

Nếu mà lỗi thuộc loại E_ERROR, chúng ta sẽ thoát chương trình và in thông báo lỗi.

```
<?php
    // void error_handler(string type, string message, string file,
    int line)
    //  Custom error handler, set by the set_error_handler()
    //  function.

    function error_handler ($type,
                            $message,
                            $file=__FILE__,
                            $line=__LINE__)
    {
      error_log("$message, $file, $line", 3, 'error_file');

      if ($type & E_ERROR) {
           print 'An error occurred, it has been logged
                 and it will be addressed.';
          exit;
      }
    }
    set_error_handler('error_handler');
?>
```

11. Lạm dụng Hướng đối tượng (HĐT)

Mô hình hướng đối tượng là một khái niệm tuyệt vời. Nó có rất nhiều lợi điểm, mà đáng chú ý nhất là khả năng dùng lại mã dễ dàng. Tuy nhiê, theo như chúng ta được hiểu: PHP không phải là một ngôn ngữ HĐT.

Mặc dù PHP có một sự hỗ trợ đầy đủ về HĐT, nó không hiệu quả lẫn không khôn ngoan nếu dùng tính năng HĐT của nó khi bạn có các hàm khác để đạt được cùng kết quả. Lí do là sự hỗ trợ HĐT của PHP không được phát triển mạnh.

Trong khi có hầu hết các phần tử chính yếu, PHP vẫn còn thiếu vài tính năng cao cấp (như các khái niệm protected, private) mà một ngôn ngữ HĐT thực sự (thí dụ như C++ , Java) phải có.

Các mã hỗ trợ HĐT của PHP không được tinh chỉnh và cũng không hiệu quả. Nghĩa là nếu bạn dùng mô hình HĐT trong PHP, bạn có thể làm chậm chương trình đáng kể.

Nói chung, một ứng dụng dùng HĐT sẽ chậm đi, cũng như là bạn dùng eval() thì sẽ chậm hơn là dùng mã bình thường. Để minh hoạ đầy đủ hơn việc HĐT có gì đó không tốt, tôi đã từng phải dùng những tính năng và khái niệm cao cấp của PHP, một vài trong số đó thậm chí chưa có tài liệu chỉ dẫn.

1. Chúng ta có thể làm gì mà không cần HĐT?

Nếu bạn chuyển sang PHP từ các ngôn ngữ như Java hay C++ (nơi bạn bạn thực sự không thể tạo các chương trình phức tạp mà không dùng các tính năng HĐT), việc bỏ qua khả năng HĐT của PHP có thể sẽ khó khăn. Dù sao, tôi vẫn có thể trấn an bạn là các chương trình rất mạnh *có thể* được viết mà không dùng mất cứ khái niệm và mô hình HĐT nào (PHP được viết bằng C, ngôn ngữ không hỗ trợ HĐT).

Để dành cho những ai không quen với kĩ năng phi-HĐT, dưới đây là vài kĩ thuật để tạo chương trình có tính kết dính và dễ mở rộng mà không dùng mô hình HĐT:

- Tạo một API
- Tạo một trình tự tên
- Nhóm các hàm liên quan vào một tập tin

1.1. Tạo một API

Áp dụng 3 lớp cho chương trình của bạn:

- Thứ nhất, các hàm thực sự thực hiện công việc của bạn
- Thứ hai, một hàm API. Đây là hàm giúp bạn xây dựng các chương trình đặc thù
- Chương trình

MortgageRate.php

```
<?php
    // The internal functions are layer 1

    // Internal function to calculate the correct
    // interest rate to be used given the amount per month
    // and the time it is to be paid in

    function _mort_find_interest_rate ($total) {
       if ($total < 30000)
           return (7.4);
       elseif ($total > 30000)
           return (3.2);
       elseif ($total > 50000)
           return (2.5);
       else
           return (1.7);
    }
    // The API is layer 2

    // double calculate_mortgage_rate (int money, int time, int month)

    //   Calculate the mortgage rate given the
    //   the total money, time its paid over and
    //   the intervals

    function calculate_mortgage_rate ($money, $time, $month) {
        $rate       = _mort_find_interest_rate ($money) / 100;
        $money     /= ($time / $month);
        return ($rate * $money) + $money;
    }
?>
```

CalcMortgage.php

```
<?php
    // The actual application is layer 3
    // $money, $time and $period are submitted
    // from a form
    include_once 'MortgageRate.php';
    $price = calculate_mortgage_rate ($money, $time, $period);
    print "Your $period month cost is $price";
?>
```

1.2. Tạo một trình tự tên và luôn tuân thủ

Một trong những vấn đề chính trong bất cứ một dự án lớn nào là sự xung đột về tên. Các lớp có thể phân đoạn tên. Do đó, các lớp khác nhau có thể:

- Được gán một thuộc tính với tên giống nhau
- Chứa các phương thức với tên giống nhau

Thí dụ, lớp Phillips và lớp Normal có thể cùng có phương thức tên screwdriver.

Nói chung, trước khi bắt đầu một dự án lớn nào, bạn nên có một trình tự tên cho mọi thứ, cụ thể là cách bạn tách các biến toàn cục ra các biến thông thường, cách định nghĩa hàm trong thư viện v.v.

1.3. Nhóm các ý niệm chung vào một tập tin

Nhóm các hàm API tương tự vào chung một tập tin cũng giống như nhóm các phương thức tương tự vào một lớp. Cố gắng tưởng tượng mỗi tập tin bạn tạo là một lớp, mỗi hàm trong đó là một phương thức. Bằng cách này, các hàm của bạn sẽ có định nghĩa và cấu trúc sáng sủa.

Thí dụ, bạn có thể muốn nhóm mọi hàm liên quan đến truy cập CSDL vào một tập tin DB.php.

2. HĐT, giống mọi thứ, tốt khi có điều độ

Để tôi làm sáng tỏ một việc. Tôi không phải đang cố biện hộ để bạn từ bỏ *hẳn* HĐT trong PHP. Đúng ra, tôi chỉ đang cố cảnh báo bạn đừng nên dùng PHP như Java hay C++, nơi mà HĐT có thể dùng thoải mái.

Hãy cẩn thận đánh giá lợi và hại trước khi bạn dùng một tiếp cận HĐT với PHP.

10. Dùng nhầm Biểu thức Chính quy

Biểu thức chính quy (Regular expressions) là công cụ mạnh để tìm và tổ chức dữ liệu, như là kiểm định địa chỉ e-mail hoặc kiểm tra một URL. Tuy nhiên, nó chậm hơn các công cụ của PHP trong một số tác vụ đơn giản.

Thí dụ, nếu bạn muốn viết hoa toàn bộ một chuỗi, một lính mới của PHP có thể làm như sau:

```
<?php
    $URL = "http://www.php.net";
    $fp = @fopen ($URL, "r");
    if (!$fp) {
        die ("Cannot open website $URL!");
    }

    while ($line = @fgets ($fp, 1024)) {
        $data .= $line;
    }

    @fclose ($fp)
    or warn ("Cannot close website handle, $URL");
    $data = ereg_replace ("[a-z]", "[A-Z]", $data);
    print $data;
?>
```

Tuy nhiên, sẽ phí thời gian khi bạn dùng hàm ereg_replace() (chậm hơn) để làm công việc mà strtoupper() (nhanh hơn) có thể thực hiện tốt hơn.

```
$data = strtoupper ($data);
```

Nói chung, bạn luôn cố gắng dùng các thay thế đơn giản của biểu thức chính quy vì nó sẽ làm tăng khá nhiều tốc độ chương trình.

1. Các hàm cần biết

Có vài hàm rất thiết yếu để tiết kiệm thời gian thi hành chương trình khi dùng thay cho biểu thức chính quy. Dưới đây là danh sách các hàm *thiết yếu* đó:

strtoupper(); strtolower(); ucfirst(); strtr(); str_replace(); trim(); explode(); implode(); substr(); strcmp()

Nếu bạn thay thế các biểu thức chính quy của bạn bằng các hàm trên, bạn có thể trông đợi một sự nhảy vọt về hiệu năng, đặc biệt khi bạn làm việc với các chuỗi lớn.

9. Lập trình PHP như các ngôn ngữ khác

Nhiều người bắt đầu PHP sau khi đã thuần thục một ngôn ngữ khác như Perl, C, Java hay ASP  . Làm như vậy, họ cũng mang theo những mô hình mà có thể không luôn luôn được dùng bởi PHP.

Không may thay, vài cá nhân trong số hày không chịu bỏ thời gian để học cách lập trình PHP theo cách phù hợp với PHP. Thay vào đó, họ thích PHP hoạt động với các ít khái niệm mới càng tốt

Khi bạn lập trình PHP như là trong các ngôn ngữ khác, nó thường dẫn đến việc làm chương trình chậm hơn và khó bảo trì mã. Bạn sẽ thường thấy họ phạm vào một trong các lỗi sau:

Perl "1 dòng"

PHP là một ngôn ngữ không thực sự tối ưu cho cách tiếp cận 1 dòng khi viết chương trình. Thay vào đó, nó mở rộng các tập hàm phức tạp và các biểu thức chính quy theo một định dạng có cấp bậc hơn

Mã lệnh Perl

```
while (<STDIN>) {
    @_ = split /:/;
    $quotes{shift} = shift;
}
print map { "$_: ", reverse split //,$quotes->{$_},"\n";} keys
%quotes;
```

Mã lệnh PHP)

```
<?php
     $fp = @fopen('php://stdin', 'r');
     if (!$fp) {
         die ('Cannot open STDIN');
     }

     while ($line = @fgets ($fp, 1024)) {
         list($name, $quote) = explode (':', $line);
         $quotes[ $name ] = $quote;
     }

     foreach ($quotes as $name => $quote) {
         print "$name: ";
         print implode (" ", array_reverse (preg_split ('//',
                        $quote)));
         print "\n";
     }

     @fclose ($fp);
?>
```

Không dùng các hàm sẵn có

Nhiều lập trình viên PHP có nền tảng C có vẻ không nhận ra rằng PHP cung cấp nhiều hàm sẵn có giúp thay thế các đoạn mã dài. Nếu bạn đến với PHP từ C, tôi khuyên bạn nên đọc qua tài liệu trước khi viết một khối lệnh để xem PHP có sẵn hàm nào giúp cuộc sống của bạn dễ dàng hơn không.

Đổi tên các hàm PHP đã có

Tôi đã thấy người ta đổi tên các hàm đã có của PHP chỉ để giúp họ dễ nhớ hơn. Điều này không chỉ làm chậm chương trình, mà làm cho đoạn mã khó đọc hơn.

Dùng hướng đối tượng quá mức

PHP không phải là một ngôn ngữ HĐT, dù nó cung cấp các tính năng HĐT. Bạn cần luôn nhận thức rằng HĐT trong PHP sẽ làm chậm đáng kể chương trình.

1. Lấy thông tin ở đâu?

May mắn là có rất nhiều thông tin về việc *làm thế nào* để lập trình PHP. Vài nơi tốt nhất là

- ~~Zend.com~~ bản gốc của bài viết này là ở đó
- ~~Professional PHP~~ một trong những sách toàn diện tốt nhất về PHP, tốt cho cả lập trình viên và không-lập trình viên.
- ~~Web Application Development with PHP~~ Một quyển sách tuyệt vời dạy bạn cả cách phát triển web lẫn vài tính năng cao cấp của PHP. Bao gồm các tài liệu chính thức về Zend API.
- ~~The PHP Developer's Cookbook~~ Một quyển sách thiên về lời giải đáp cho các vấn đề gặp phải trong PHP (do tui - Sterling Hughes - viết cùng Andrei Zmievski)

8. Không nhận thức đầy đủ về bảo mật

Những người dùng không phải lúc nào cũng làm việc với hệ thống của chúng ta. Với tư cách là lập trình viên, trách nhiệm *của chúng ta* là thiết kế một hệ thống an toàn, dễ chịu có thể làm việc được chung với lỗi của người dùng.

Khi thiết kế hệ thống, bạn phải đặt mình vào vị trí người dùng. Xem xét những chỗ họ có thể gặp lỗi và tìm kiếm những lỗ hổng bảo mật tiềm tàng. Rồi bạn thiết kế chương trình có khả năng sửa chữa lỗi này và lấp các lỗ hổng bảo mật. Một điều khác cũng quan trọng là: dù xảy ra hư hỏng hay hệ thống bị tấn công là do lỗi của người dùng, *chính bạn* là người chịu trách nhiệm nếu bạn đã viết chương trình có nhiều lỗi hay thiếu những bước kiểm tra cần thiết dẫn đến hỏng dữ liệu.

Thí dụ, tôi đã thấy nhiều chương trình không dùng hàm đã có của PHP mail() vốn an toàn mà lại đi dùng sendmail thông qua popen(). Nó có thể dẫn đến nhiều lỗ hổng bảo mật (thí dụ như /etc/passwd được gửi đến người dùng cuối).

Có vài nơi thường xảy ra các sự cố về bảo mật, hoặc tiềm năng gây ra hỏng dữ liệu rất lớn:

- *Thực hiện lời gọi hệ thống. Tôi không có cách nào diễn tả được mức độ nguy hiểm của vấn đề này. Luôn đảm bảo rằng mọi dữ liệu người dùng đưa vào là an toàn trước khi chuyển nó cho một lời gọi hệ thống.* ***ĐỪNG BAO GIỜ TIN NGƯỜI DÙNG MỘT CÁCH MÙ QUÁNG KHI ĐƯA DỮ LIỆU CỦA HỌ VÀO LỜI GỌI HỆ THỐNG MÀ KHÔNG KIỂM TRA TRƯỚC.***
- *Khi đăng kí người dùng. Nếu bạn trông mong có kết quả chính xác, luôn kiểm tra và đảm bảo bản đăng kí được kiểm tra những chỗ cần thiết. Trước tiên, có nhiều cách để kiểm tra một địa chỉ e-mail hợp lệ. Hơn nữa, bạn nên kiểm tra tuổi của người dùng có nằm trong phạm vi thích hợp không. Với một sự suy rộng, bạn có*

thể khẳng định khá chắc chắn rằng không có một người 200 tuổi nào có khả năng dùng máy vi tính.

- *Khi chấp nhận thẻ tín dụng. Một số lập trình viên chỉ dùng những thuật toán đơn giản mà có thể dễ dàng bị đánh lừa khi thực hiện kiểm tra thẻ tín dụng. Chỉ chấp nhận thẻ của các công ty lớn để kiểm tra tính hợp lệ của số thẻ trước khi chấp nhận thẻ của người dùng.* ***ĐỪNG BAO GIỜ TIN TƯỞNG MỘT THUẬT TOÁN***

1. Bảo mật của lời gọi hệ thống

Mỗi khi bạn đưa dữ liệu của người dùng vào lời gọi hệ thống, bạn cần *tỉnh táo kiểm tra* dữ liệu đó. Đảm bảo rằng không có gì nguy hiểm nằm trong dữ liệu đó có thể lừa phỉnh hệ thống thực hiện những lệnh không mong muốn. PHP cung cấp một hàm làm điều đó: EscapeShellCmd()

Bất cứ khi nào bạn chuyển một lệnh có chứa dữ liệu nhạy cảm, trốn thoát dữ liệu đó bằng hàm **EscapeShellCmd**():

Trốn thoát (escaping) dữ liệu có nghĩa là thêm dấu sổ ngược (backslash \) trước kí tự có thể lừa phỉnh hệ thống (chính xác là các kí tự #&;?'\"|*?~<>^()[]{}$\\\x0A\xFF).

Mã HTML

```
<html>
<head>
<title>Name Lookup</title>
</head>
<body>
    <h1>Name Lookup</h1>
    <?php
        if ($name) {
           system (EscapeShellCmd ("lookup $name"));
           print "
           nn";
        }
    ?>
    <form action="<[color=blue]?php print $PHP_SELF; ?>[/color]"
    method="GET">
        Enter a name to lookup:
        <input type="text" name="name">
        <input type="submit" value="Lookup Name">
    </form>
</body>
</html>
```

Dù EscapeShellCmd() là một hàm tốt để kiểm tra lệnh, bạn vẫn nên thử và thực hiện các kiểm tra đặc thù phụ thuộc vào loại dữ liệu. Hàm EscapeShellCmd() sẽ không kiểm tra tính đúng đắn của dữ liệu được đệ trình, nó sẽ chỉ ngăn cản người dùng làm các việc không được phép.

1.1. Đi xa hơn một bước

Như một quy luật, nên kiểm tra các kí tự *được phép* hơn là kiểm các kí tự *không được phép*.

Thí dụ, đảm bảo rằng $name chỉ chứa các kí tự chữ và số (alphanumeric characters). Bằng cách này, rất khó để khai thác lỗ hổng trong hệ thống của bạn.

2. Kiểm tra địa chỉ e-mail

Một trong những hình thức kiểm tra phổ biến nhất là xem một địa chỉ e-mail có hợp lệ không. Mấy tay mới vào nghề sẽ chỉ dùng các biểu thức chính quy (mà họ lượm được trong

một nhóm thư tín, hay trong một kho mã nguồn nào đó). Tuy nhiên, một biểu thức chính quy không đủ nếu bạn muốn có kết quả chính xác. Có vài cách an toàn hơn mà bạn có thể dùng:

2.1. Kiểm tra kết nối (socket validation)

Một cách để kiểm định địa chỉ e-mail mà không quấy rầy trực tiếp đến người dùng là tạo một kết nối đến server nhận được trong địa chỉ e-mail, sau đó tìm tên đăng kí của họ.

2.1.1. Ưu điểm

- Không gây bất tiện cho người dùng vì nó được tiến hành trong suốt.
- Phát hiện nhiều địa chỉ ma mà biểu thức chính quy không biết (như là joe@fgsdh.com).

2.1.2. Khuyết điểm

- Không phát hiện được địa chỉ bị đánh cắp. Thí dụ, nếu John Doe đưa địa chỉ e-mail của tôi (phanthanhkieu@php.net), mọi thứ vẫn tiến hành bình thường, dù nó là địa chỉ của tôi chứ không phải của hắn ta.
- Kiểm tra chậm hơn biểu thức chính quy.
- Mail server của người dùng có thể tạm ngưng hoạt động trong thời gian ngắn, làm cho một địa chỉ e-mail hợp lệ bị từ chối

2.2. Kiểm tra tương tác

Một cách khác để kiểm định địa chỉ e-mail là gửi một khoá đặc biệt đến hộp thư người dùng, và bắt họ nhập khoá đó để tiếp tục. Điều này đảm bảo rằng không những địa chỉ e-mail là hợp lệ, mà người dùng có quyền truy xuất vào địa chỉ đó.

2.2.1. Ưu điểm

- Là cách tốt nhất để kiểm tra người dùng có một địa chỉ e-mail hợp lệ

2.2.2. Khuyết điểm

- Đòi hỏi người dùng phải tiến hành thêm bước phụ. Điều này quấy rầy mấy người có ý định làm phiền bạn.
- Giống như mọi cách khác, nó không đảm bảo được chống-đánh-lừa. Người dùng có thể tạo một địa chỉ tạm ở Hotmail hay Netaddress và sau

PHẦN 3: 7 LỖI CHẾT NGƯỜI

7. Chép và dán: không phải hướng đi tốt

Tôi đã gặp nhiều tay lập trình mới vào nghề đi chép những đoạn mã (thí dụ như là kiểm tra địa chỉ e-mail, gửi e-mail, lấy giá trị từ form để soạn e-mail). Họ thường dán chúng vào chương trình của mình và rồi kết quả là một đống những câu lệnh gửi form không an toàn.

Dù rằng một đoạn mã chạy được trong những điều kiện tối ưu, nó thường gặp lỗi trong bất kì sự kiểm tra "mã tốt" nào Công việc chắp vá sẽ **không** có các ưu điểm sau:

- ***Mở rộng được:*** *đoạn mã trông như các mảnh riêng rẽ ráp vá vào nhau. Nếu nhờ một lập trình viên có kinh nghiệm sửa chương trình của bạn, họ thường thích viết lại toàn bộ Đoạn mã không đọc được sẽ không mở rộng được.*
- ***An toàn:*** *bạn có thể đang chép mã của người khác vào chương trình của mình mà lại không hiểu rõ ràng đoạn mã đó. Hãy tưởng tưởng xem. Nếu đoạn mã đó có một lỗi làm xoá toàn bộ đĩa của bạn Hơn nữa, cùng một đoạn mã thì tính an toàn không giống nhau trên các hệ thống khác nhau. Cuối cùng, chương trình của bạn sẽ kế thừa lỗi của người khác.*
- ***Nhanh:*** *khi cắt dán các đoạn mã, kết quả thường sẽ chạy không nhanh, vì chúng không có một tiến trình chung hợp lí (đây là điều quan trọng nhất khi viết các chương trình nhanh)*

1. Làm đúng phương pháp: tìm hiểu trước, sao chép sau

Nghiên cứu mã của người khác kĩ lưỡng trước khi sao chép. Phân tích xem nó làm gì. Chỉ khi nào đoạn mã ấy *đọc được*, nhất quán với logic chương trình của bạn, và không có lỗi, thì mới nên sao chép nó. Tiến hành lúc này sẽ giúp bạn chỉnh sửa nó phù hợp với các phần còn lại của chương trình một cách thoải mái hơn.

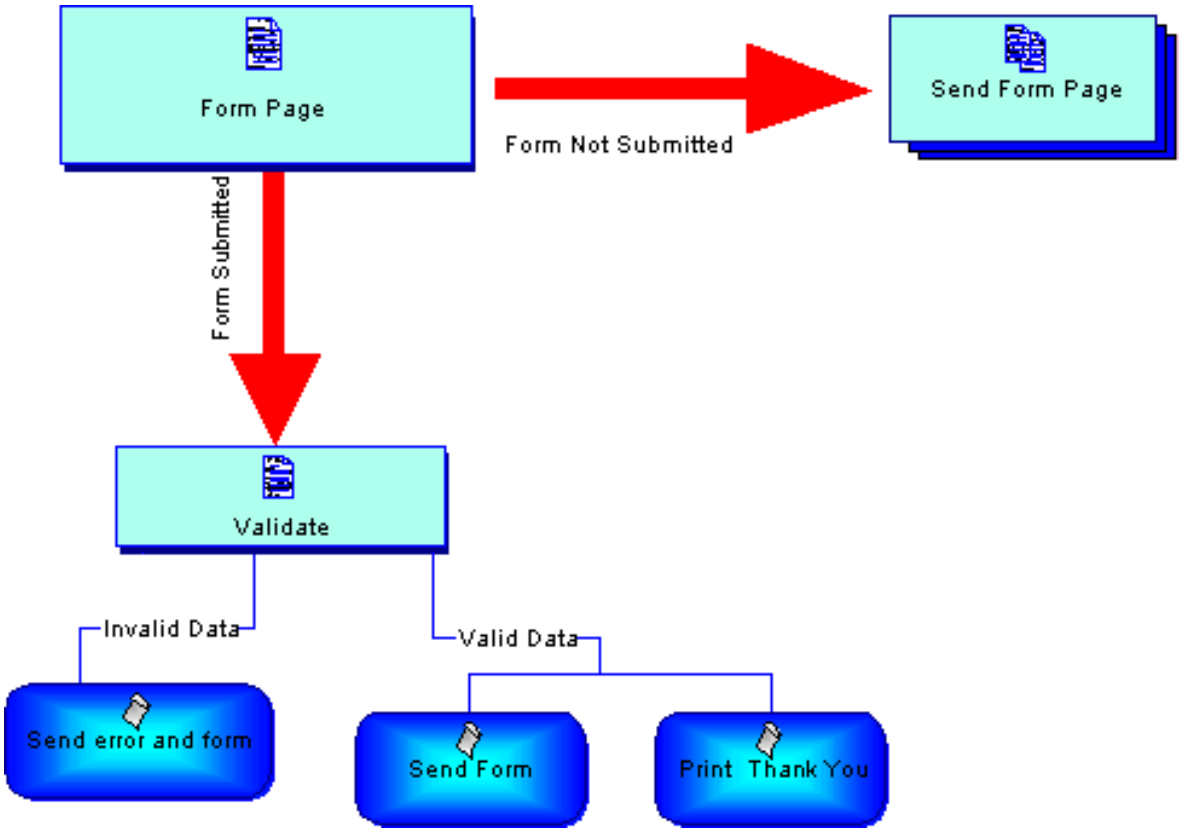
2. Các thư viện rất tốt

Chỉ dùng các thư viện PHP từ các nguồn tin cậy như PEAR hoặc kho các lớp của PHP (PHP Classes Repository). Với các gói API có sẵn, việc dùng các hàm của nó cũng không thành vấn đề. Thực ra, nếu bạn tìm được một thư viện viết sẵn bởi một nguồn tin cậy, thì thường nó sẽ là một lựa chọn tốt để dùng cho chương trình (thay vì tự viết lấy).

6. Không có đường lối (guidelines) cho đồ án

Một lần khi tôi mới bắt đầu lập trình, tôi làm việc trên một đồ án khá cơ bản (với Perl) cùng 3 lập tình viên khác. Vì tôi còn trẻ (và không phải trưởng nhóm), chúng tôi không có đường lối viết mã cho đồ án. Mỗi người được giao một phần công việc, và làm riêng rẽ. Khi chúng tôi nhập chung lại để chạy chương trình cuối cùng, mỗi phần của đồ án nhìn khác hẳn nhau

1. Một thí dụ mẫu về đường lối



5. Không xem xét lại mã
4. Vá lỗi khi thiết kế
3. Không quan tâm đến người dùng khi thiết kế
2. Không theo sát kế hoạch của đồ án
1. Phí thời gian