

PHP 变量命名规则

- 1、变量以美元符号\$开头。如\$name, \$age。
- 2、美元符号\$后面的第一个字符不可以是数字，只能是下划线_或者字母。如\$1_1 这样的变量是错误的。
- 3、除了下划线_外，变量不允许出现任何空格或标点符号。也就是说变量名只能包含：a-z、A-Z、0-9 以及下划线_。
- 4、PHP 变量名是区分大小写的。如\$name 与\$Name 是两个不同的变量。

常量的定义：

```
define("常量名", 常量值 );  
const 常量名 = 常量值;  
const 语法只能用在“顶层代码”位置，而不能用在大括号的位置，比如：  
const c1 = 1; //ok  
if ( $v1 > 5){  
    const c2 = 2;    //错误的!!!  
}
```

常量的使用——取值：

方法 1：直接使用常量名：

```
echo C1;
```

方法 2：使用函数 constant() 取值：

```
echo constant("C1");
```

这种方式有灵活性：

```
$i = 1;
```

```
$s = "C" . $i;
```

```
echo constant($s);    //这里也是取得 C1 这个常量的值；
```

常量变量的区别

判断常量是否存在：

```
$v1 = defined("C1"); //true;  
$v2 = defined("CCC222"); //false
```

常见的其实是这样：

```
if( defined(常量名) ){  
    .....  
}  
else{  
    .....  
}
```

使用一个未定义的常量：

需要注意而已！它会当做有该常量一样去使用，其值就是该名字。

预定义常量：

```
PHP_OS  
PHP_VERSION  
PHP_INT_MAX  
M_PI
```

魔术常量：目前就 3 个

```
__DIR__  
__FILE__  
__LINE__
```



数据类型

总体划分

基本类型（标量类型）：int, float, string, bool

复合类型：array, object

特殊类型：null, resource

整数类型

整数类型的 4 种写法：

\$n1 = 123;

\$n2 = 0123; //以 0 开头的数字就是 8 进制形式

\$n3 = 0x123; //以 0x 开头的数字是 16 进制

\$n4 = 0b101010; //二进制

整数类型

整数类型的 4 种写法:

```
$n1 = 123;  
$n2 = 0123; //以 0 开头的数字就是 8 进制形式  
$n3 = 0x123; //以 0x 开头的数字是 16 进制  
$n4 = 0b101010; //二进制
```

进制转换问题

4 个单词: dec, bin, oct, hex

进制转换的系统函数——必须会用

10 进制转换为 2,8,16 进制: 要求给出的原始数据是一个 10 进制的数字——如果不是, 就会自动转换为 10 进制数字。结果, 返回的是一个对应进制数字的字符串。

```
decbin($n);  
decoct($n);  
dechex($n)
```

2,8,16 进制转换为 10 进制: 要求给出的原始数据是一个应进制数字的字符串, ——如果不是, 就会自动转换为字符串。结果, 返回的是一个 10 进制的数字值;

```
$v1 = bindec($n);  
$v2 = octdec($n);  
$v3 = hexdec($n)
```

进制转换的人工计算——了解其原理

2,8,16 进制, 转换为 10 进制的基本做法: 将每位数字乘以该位的权值, 然后相加的结果

10 进制, 转换为 2,8,16 进制的基本做法: 除 2,8,16, 取余, 直到商为 0, 所有余数倒着写出

浮点类型

浮点数的 2 种表示形式

有小数点: \$f1 = 1.23;

有科学计数法的符号 E: \$f2 = 1.23E4; \$f2 = 123E4; \$f2 = 123E4def;

错误, 无法识别

浮点数使用的细节知识

浮点类型

浮点数的 2 种表示形式

有小数点: \$f1 = 1.23;

有科学计数法的符号 E: \$f2 = 1.23E4; \$f2 = 123E4;

浮点数使用的细节知识:

1, 不要对浮点数直接进行大小比较, 而应该在考虑应用的精度需求的基础上, 转换为整数然后进行比较。

2, (了解): 浮点数小数的二进制转换: 将小数部分乘以 2, 取得其整数部分, 如果还有小数, 继续乘以 2, 直到小数消失 (其实极少会发生这种情况), 然后将所有整数部分按顺序写出来, 就是对应的小数。

3, 整数运算如果超过了整数的最大值, 会自动转换为浮点数。|

作业:

```
23>>转为16进制:
```

```
23
```

```
/16
```

```
=1      7
```

```
/16
```

```
=0      1
```

```
结果就是17
```

```
234>>转为16进制:
```

```
234
```

```
/16
```

```
=14     A
```

```
/16
```

```
=0      E
```

```
结果就是EA
```

运算符

算数运算符

基础:

符号有: + - * / %

说明:

- 1, 他们都是针对数字进行的运算:
- 2, 如果他们的两边有不是数字的数据, 就会自动转换为数字
- 3, 其中取余运算(取模运算)% , 它只针对整数进行运算, 如果不是, 会自动截取为整数。

11.3 % 3 相当于 11 % 3

11.8 % 3.8 相当于 11 % 3

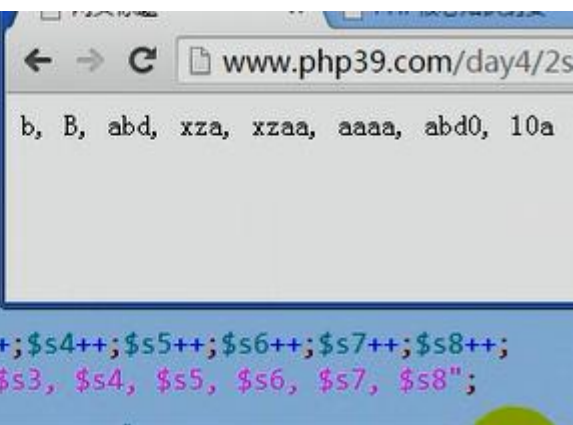
自增自减运算符:

- 常规: 对数字进行自加 1 或自减 1。

- 字符串：只能自增，且自增的效果就是“下一个字符且只能针对字母或数字进行自加”
- 布尔值递增递减无效
- null 递减无效，递增结果为 1

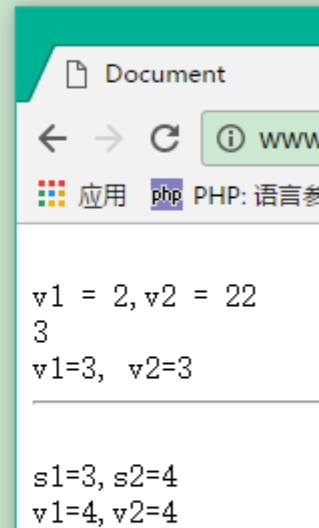
字符串自增的例子：

```
13 <?php
14 $s1 = "a";
15 $s2 = "A";
16 $s3 = "abc";
17 $s4 = "xyz";
18 $s5 = "xyzz";
19 $s6 = "zzz";
20 $s7 = "abc9";
21 $s8 = "9z";
22 $s1++;$s2++;$s3++;$s4++;$s5++;$s6++;$s7++;$s8++;
23 echo "$s1, $s2, $s3, $s4, $s5, $s6, $s7, $s8";
24 ?>
```



前自增和后自增的区别（自减类似）

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5 <meta name="Generator" content="EditPlus®">
6 <meta name="Author" content="">
7 <meta name="Keywords" content="">
8 <meta name="Description" content="">
9 <title>Document</title>
10 </head>
11 <body>
12 <?php
13 $v1 = 1;
14 $v2 = 1;
15 $v1++; //此行后, v1为2
16 ++$v2; //此行后, v2为2
17 echo "<br />v1 = $v1,v2 = $v2";
18 //说明: 此时前自增后自增效果一样!
19
20 echo $v1++; //输出2, 此行后, v1为3
21 echo "<br />";
22 echo ++$v2; //输出3, 此行后, v2为3
23 echo"<br />v1=$v1, v2=$v2";
24
25 echo "<hr />";
26 $s1 = $v1++; //s1为3, 此行后, v1为4
27 $s2 = ++$v2; //s2为4, 此行后, v1为4
28 echo "<br />s1=$s1,s2=$s2";
29 echo "<br />v1=$v1,v2=$v2";
30 //可见, 在有加加运算的其他语句中,
31 //前加加和后加加会有区别
32 //影响其他语句的执行结果
33 //前加加是先对自己变量加1, 然后做其他运算
34 //后加加是先做其他运算, 然后对自加变量加1
35
36 ?>
37 </body>
38 </html>
39
```



通常, 我们在循环中, 推荐使用, 前加加, 比如:

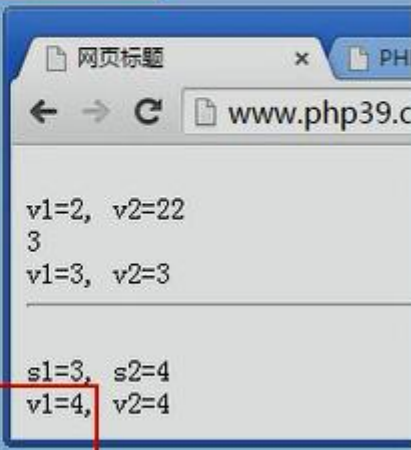
```
For($i = 1; $i<10000; ++$i){.....}
```

演示前加加后加加进行 1 千万次的“效率比较”:


```

14 $v1 = 1;
15 $v2 = 1;
16 $v1++; //此行后，v1为2
17 ++$v2; //此行后，v2为2
18 echo "<br />v1=$v1, v2=$v2";
19 //说明：独立的加加运算中，前自增后自增效果一样！
20
21 echo $v1++; //输出2，此行后，v1为3
22 echo "<br />";
23 echo ++$v2; //输出3，此行后，v2为3
24 echo "<br />v1=$v1, v2=$v2";
25
26 echo "<hr />";
27 $s1 = $v1++; //s1为3，此行后，v1为4
28 $s2 = ++$v2; //s2为4，此行后，v2为4
29 echo "<br />s1=$s1, s2=$s2";
30 echo "<br />v1=$v1, v2=$v2";
31 //可见，在有加加运算的其他语句中，
32 //前加加和后加加会有区别：
33 //影响其他语句的执行结果：
34 //前加加是先对自加变量加1，然后做其他运算
35 //后加加是先做其他运算，然后对自加变量加1

```



比较预算符

符号：> >= < <= == != === !==

- 一般比较：是针对数字进行的大小比较
- ==和===比较：前者通常叫做模糊比较，后者叫做精确相等的比较（只有数据的类型和数据的值/内容，都相等，才是全等的）。必须能够找到手册的“类型比较表”：附录》php 类型比较表

松散比较 ==												
	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE

严格比较 ===												
	TRUE	FALSE	1	0	-1	"1"	"0"	"-1"	NULL	array()	"php"	""
TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE

- 不要对浮点数直接进行大小比较

常见不同类型（标量类型）数据之间的比较规律：

- 如果比较的数据中，有布尔值，转为布尔值比较，布尔值比较只有一个规则：`true > false`
- 否则，如果有数字值，就转为数字值比较：这就是常规比较。
- 否则，如果两边都是“纯数字字符串”，转为数字比较
- 否则，就按字符串比较，字符串表的规则是：
 - 对两边的字符串，一个一个从前往后取出字符并进行比较，谁“先大”，结果就是它大。

```

“abc” > true    //? False
“abc” > false   //true
“0” > false     //false
3 > “12”        //false
3 > “12abc”     //false
“3” > “12”     //false
“abc” > “c” ;   //false 后者大
“abc” > “ab123cde” ; //true 因为这里“c”大于“1”
“3abc” > “12abc” // true, 因为字符串“3” > 字符串“1”

```

逻辑运算符

逻辑运算符都是针对“布尔值”进行的运算

如果不是布尔值，就会转换为布尔值进行运算

布尔值只有 2 个：true, false

基本运算规则（真值表）

逻辑与规则：

true && true → true

false && true → false

true && false → false

false && false → false

总结：只有 2 个都是 TRUE，结果才是 true

只要有 1 个 FALSE，结果就是 false

逻辑或规则：

true || true → true

false || true → true

true || false → true

false || false → false

总结：只有 2 个都是 false，结果才是 false

只要有 1 个 true，结果就是 true

逻辑非规则：

!true → false

! false → true

逻辑运算符的“短路现象”：

逻辑与短路

```
18     $n1 = 3;
19     $n2 = 4;
20     //if判断语句写法1:
21     if($n1 > $n2 && f1($n1,$n2) > 20){
22         //完成某种任务1
23     }
24     else{
25         //完成某种任务2
26     }
27     //if判断语句写法2:
28     if(f1($n1,$n2) > 20 && $n1 > $n2 ){
29         //完成某种任务1
30     }
31     else{
32         //完成某种任务2
33     }
34     //写法1和写法2，最终计算结果是一样的！
35     //但写法1会具有优势：它有时候可能不需要进行“复杂”计算
36     //就可以得到判断结果，这就是“短路”
37     //而写法2却总是先去进行“复杂”计算，显然属于消耗资源行为
38
```

结果: 如果一个语句中，通过与运算需要进行多项判断，而且不同的判断具有明显不同的“复杂程度”，则我们应该将简单的判断放在前面，这时候我们就可以利用短路现象以达到提高效率的目的

逻辑或短路

```

43 //函数f1表示一个复杂的计算:
44
45 $n1 = 3;
46 $n2 = 2;
47 //if判断语句写法1:
48 if( $n1 > $n2 || f1($n1, $n2) > 20 ) {
49     //这里完成某种任务1
50 }
51 else{
52     //这里完成另一些任务2
53 }
54 //if判断语句写法2:
55 if( f1($n1, $n2) > 20 || $n1 > $n2 ) {
56     //这里完成某种任务1
57 }
58 else{
59     //这里完成另一些任务2
60 }
61 //写法1和写法2，最终计算结果是一样的！
62 //但写法1会具有优势：它有时候可能不需要进行“复杂”计算
63 //就可以得到判断结果，这就是“短路”现象
64 //而写法2却总是先去进行“复杂”计算，显然属于消耗资源行为
65

```

简单运算，应该放在前面

结果: 如果一个语句中，通过与运算需要进行多项判断，而且不同的判断具有明显不同的“复杂程度”，则我们应该将简单的判断放在前面，这时候我们就可以利用短路现象以达到提高效率的目的

字符串运算符

- 1, 符号只有一个: `.` 也衍生出另一个: `.=`
- 2, 含义: 就是将这个符号两边的字符串链接起来;
- 3, 如果两边不是字符串, 就会自动转换为字符串, 然后连接起来。

“ab . 3 ➔ ”ab3” ;
 “12” . 3 ➔ ”123”
 12 . 3 ➔ ”123”

赋值运算符:

一个基本赋值运算符: `=`

形式: `$变量名 = 值;`

理解: 将右边的值 (不管做了多少运算), 赋值给左边的变量。

若干个衍生的赋值运算符:

`+=` 加等：形式： `$变量名 += 值;`
理解： 相当于： `$变量名 = $变量名 + 值;`

`-=` 减等：形式： `$变量名 -= 值;`
理解： 相当于： `$变量名 = $变量名 - 值;`

`*=` `/=` `%=` `.=` 其都可以认为是上述形式的一种简化版。

条件（三目，三元）运算符

只有一个，形式如下：

数据值 1 ? 数据值 2 : 数据值 3

含义：

对数据值 1 进行判断，如果为“真”，则该运算符的运算结果就是数据值 2，否则就是数据值 3；它是这样一个流程控制（逻辑判断）语句的简写形式：

```
If( 数据值 1 (一般是变量) ){
```

```
    $变量名 = 数据值 2;
```

```
}
```

```
else{
```

```
    $变量名 = 数据值 3;
```

```
}
```

注意：如果数据值 1 不是布尔值，也会转换为布尔值；

```
$score = 66;    //分数
```

```
$valuation = $score >= 60? “及格” : “不及格”; //结果为“及格”
```

```
$score = 56;    //分数
```

```
$valuation = $score >= 60? “及格” : “不及格”; //结果为“不及格”
```

```
$score = 56;    //分数
```

```
$valuation = $score ? “及格” : “不及格”; //结果为“及格”，这里可能就偏离了本意!!
```

位运算符

基础规定

- 1，位是什么？就是 2 进制数字的每一个“位”，一个整数数字，有（由）32 个位构成！
- 2，位运算符是仅仅针对整数进行的运算符；
- 3，位运算符有如下几个

`&`：按位与；

`|`：按位或；

~: 按位非; 按位取反

^: 按位异或;

4, 位运算符的基本语法规则:

按位与基本规则:

1 & 1 → 1

1 & 0 → 0

0 & 1 → 0

0 & 0 → 0

按位或基本规则:

1 | 1 → 1

1 | 0 → 1

0 | 1 → 1

0 | 0 → 0

按位非基本规则:

~1 → 0

~0 → 1

按位异或基本规则:

1 ^ 1 → 0

1 ^ 0 → 1

0 ^ 1 → 1

0 ^ 0 → 0

可见, 按位异或的规则是: 相同为 0, 不同为 1

整数的按位与运算 (&):

形式:

\$n1 & \$n2; //n1, n2 是 2 个任意整数;

含义:

将该 2 个整数的二进制数字形式 (注意, 都是 32 位) 的每一个对应位上的数字进行基本按位与运算之后的结果!

注意: 他们运算的结果, 其实仍然是一个普通的数字 (10 进制)。

示例图示 (只用 8 个位来演示):

\$v1 = 10 & 20;

\$v1 = 10 & 20;								
10 的 2 进制	0	0	0	0	1	0	1	0
20 的 2 进制	0	0	0	1	0	1	0	0
& 运算结果:	0	0	0	0	0	0	0	0

代码验证：

```
11 <body>
12 <?php
13     $v1 = 10 & 20;
14     echo "<br />v1 = $v1";
15 >
16 </body>
17 </html>
```

v1 = 0

整数的按位或运算：

形式：

`$n1 | $n2;` //n1,n2 是 2 个任意整数；

含义：

将该 2 个整数的二进制数字形式（注意，都是 32 位）的每一个对应位上的数字进行基本按位或运算之后的结果！

注意：他们运算的结果，其实仍然是一个普通的数字（10 进制）。

示例图示（只用 8 个位来演示）：

`$v1 = 10 | 20;`

\$v1 = 10 20;								
10 的 2 进制	0	0	0	0	1	0	1	0
20 的 2 进制	0	0	0	1	0	1	0	0
运算结果：	0	0	0	1	1	1	1	0

则结果该数据值大小为： $1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 0 = 16+8+4+2=30$

代码验证：

```
16
17     $v1 = 10 | 20;
18     echo "<br />v1 = $v1";
```

v1 = 30

整数的按位左移运算（<<）：

形式：

`$n1 << $m`

含义：

将数字 n1 的二进制数字形式（也是 32 位的），的每一个位上的数字都一次性往左边 m 位，并将右边空出来的位置补 0，左边冒出去的不管，这样操作之后得到的结果。

示例图示（只用 8 个位来演示）：

```
$v1 = 10 << 2;
```

\$r1 = 10 << 2;

10 的 2 进制	0	0	0	0	1	0	1	0
左移 2 位后	0	0	1	0	1	0	0	0
则结果为：			2 ⁵	0	2 ³			

可见，结果为：2⁵ + 2³ = 32 + 8 = 40

代码验证

```
19 $v1 = 10 << 2;
20 echo "<br />v1 = $v1";
21
```

v1 = 40

可见左移几位就是乘以 2 的几次方
右移动就是除以 2 的几次方，
此方法比乘除的运算效率高，

