# 基本语法

## PHP标记

起始标记：<?php

结束标记：?>

这样写可以将php代码嵌入到任何的文档中

PHP也有短标记<? ?>,不鼓励；只有在php.ini中打开short\_open\_tag或者在编译是使用配置选项



如果文件内容是纯PHP代码，最好在文件末尾删除结束标记

版本5.4以后，增加了段标记：<?=

每个语句后面要用分号结束，如果是最后一行可以省略分号。

//最后一行可以不用使用分号，但要有php的结束标记

<?php

phpinfo()

?>

//如果没有结束标记，需要分号

<?php

phpinfo();

为什么不需要文件末尾的PHP结束标记：

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](mk:@MSITStore:F:\Html\mannual_files_all\enhancingProgram\PHP\php_manual_zh_review.chm::/res/function.include.html) 或者 [require](mk:@MSITStore:F:\Html\mannual_files_all\enhancingProgram\PHP\php_manual_zh_review.chm::/res/function.require.html) 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。

单行注释：// 或者 #

多行注释:/\* \*/,不能嵌套

PHP支持8中原始数据类型：

Boolean integer float/double string

Array object

Resource Null

伪类型：Mixed number callback

PHP中变量的类型通常不是由程序员设定的，是由PHP根据该变量使用的上下文在运行时决定的。

查看类型：var\_dump 或者 gettype 或者判断类型is\_null is\_string等

<?php

$str = 'abc';

$num = 12;

$nil = null;

var\_dump($str);

var\_dump($num);

var\_dump($nil);

echo gettype($str) . "\n";

echo gettype($num) . "\n";

echo gettype($nil) . "\n";

var\_dump(is\_int($str)) . "\n";

类型强制转换：settype 和各种前缀（int bool float string array object unset-转换为null）

<?php

//$str='addaa'; #强制转换为0

$str='12';

if(settype($str,'integer')){

var\_dump($str);

}else{

echo '强制转换失败';

}

### Boolean类型：

True和false关键字不区分大小写

<?php

$bol = TRUE;

$bol = false;

$bol = truE;

var\_dump($bol);

当运算符、函数或者控制流结构需要一个boolean参数时，会自动将参数转换为bool类型（可能是强制转换），以下值转换为boolean时，会被认为是flase：

false 0 0.0 ‘0’ 空字符串 空数组 不包含任何成员变量的对象（仅对PHP4.0适用） null 尚未赋值的变量 从空标记生成的simpleXML对象。

其他值都被认为是Ture，任何资源都为true

<?php

var\_dump((bool)"");

var\_dump((bool)"0");

var\_dump((bool)array());

var\_dump((bool)0);

var\_dump((bool)0.0);

var\_dump((bool)null);

### integer

整型值可以使用十进制、十六进制、八进制、二进制。在前面可以加上正负号。二进制自PHP 5.4.0起可用。八进制：数字前加0，十六进制：数字前加0x，二进制：数字前加0b

<?php

$a = 12; #十进制

var\_dump($a);

$b = 012; #八进制

var\_dump($b); //输出10，即输出时都以十进制表示

$c = 0x12; #十六进制

var\_dump($c);

$d = 0b11; #二进制

var\_dump($d);

$e = +12; #正数

var\_dump($e);

$f = -12; #负数

var\_dump($f);

整型数的字长和平台有关（不同的位数计算机表示的范围可能不同），PHP不支持无符号整数，自PHP 4.4.0he PHP 5.0.5后，可以使用PHP\_INT\_MAX来访问最大值。

echo 'PHP\_INT\_MAX: ' . PHP\_INT\_MAX;

如果一个整型数的范围超过了integer的范围，就会被解释为float

PHP中除法可以强制转换为integer或者使用函数round四舍五入。true转为1，false转为0；float取整时向下取整。

var\_dump(12/3); //int(4)

var\_dump(12/5); //double(2.4)

var\_dump((int)12/7); //double() 强制转换后为double

var\_dump(round(12/7)); //double(2)

var\_dump((int)true); // 1

var\_dump((int)false); // 0

不要将未知的分数强制转换为integer,如echo (int)((0.1+0.7)\*10);输出7

### float浮点型

浮点型也叫做浮点数float,双精度double或实数real，可以使用以下的方法来定义：

<?php

$a = 1.234;

var\_dump($a); //double(1.234)

$b = 1.2e+3;

var\_dump($b); //double(1200)

$d=1.2e3;

var\_dump($d); //double(1200)

$c=7e-2;

var\_dump($c); //double(0.07)

比较浮点数的两个方法：

直接比较两个浮点数时可能会有问题；如果要比较两个浮点数，需要使用一个最小误差值，该值也被称为机器极小值或最小单元取整数，作为判断两个浮点数是否相等时的允许误差。

1、直接比较

<?php

$a=1.23456781230;

$b = 1.23456781230;

var\_dump($a==$b);

2、设置允许误差

$a=1.23456781230;

$b = 1.23456781231;

var\_dump(abs($a-$b)<0.00001); //认为差值小于0.00001的两个数是相等的

NaN：not a number，代表着一个在浮点数运算中未定义或不可表述的值，与任何值的比较都是false，应该用is\_nan来判断。

var\_dump(is\_nan($a));

### String 字符串

PHP中每个字符就等同于一个字节（无论是汉字还是字母等符号），这意味着PHP只能支持256中字符集，从而不支持unicode

字符串的四种表达方式：

1. 单引号 2、双引号 3、heredoc结构 4、nowdoc结构

单引号：

在单引号中的变量和特殊字符的序列的转义将不会被替换

<?php

$a='abc';

$b='def';

var\_dump($a);

var\_dump('\t'); #这里的\t被当做两个字符

var\_dump("\t"); #这里的\t被当做一个字符-制表符

var\_dump('$b'); #这里直接输出$b

var\_dump("$b"); #这里输出def

双引号：

在双引号中，php将对一些特殊的字符进行解析，如\t、\n 等和变量；而且用双引号中的变量会被解析。

<?php

$a='abc';

$b='def';

var\_dump($a);

var\_dump('\t'); #这里的\t被当做两个字符

var\_dump("\t"); #这里的\t被当做一个字符-制表符

var\_dump('$b'); #这里直接输出$b

var\_dump("$b"); #这里输出def

heredoc结构：<<<标识符，标识符的定义规则与变量的定义规则一样，只能包含字母数字和下划线，并且必须以字母和下划线开头。

heredoc结构就像是没有使用双引号的双引号字符串，在其中的单引号不用被转义，变量将被替换。

<?php

$a='abc';

$b='def';

$c=<<<eof

this is a heredoc\tstruct

the 'first' variable $a ,

the second variable $b .

eof;

echo $c; //其中的$a和$b都被解析了,\t被解析了，‘没有被转义

自PHP 5.3.0起，还可以在heredoc结构中使用双引号来声明标识符，如：

<?php

$a='abc';

$b='def';

$c=<<<"eof"

this "is" a heredoc\tstruct

the 'first' variable $a ,

the second variable $b .

eof;

echo $c; //其中的$a和$b都被解析了,\t被解析了，‘没有被转义

nowdoc结构

就像heredoc结构类似于双引号字符串，nowdoc结构类似于单引号字符串。nowdoc中不进行解析操作（以下代码中的变量不会被解析，原样显示），结构表示法与heredoc大致相同，但标识符使用单引号括起来。

<?php

class foo{

public $foo;

public $bar;

public function foo(){

$this->foo='foo';

$this->bar=['bar1', 'bar2', 'bar3'];

}

}

$foo=new foo();

$name = 'myname';

$str=<<<’eof’

this is a class and the variatbles are

the first variable:$foo->foo

the second variale:$foo->bar

the variables in foo class are above!.

eof;

echo $str;

变量解析：

两种语法规则 1、简单语法 2、复杂语法

简单语法：当PHP解析器遇到一个美元符号$时，会和其他很多解析器一样，去组合尽可能多的表示以形成一个合法的变量名，可以使用花括号来明确变量名边界。

<?php

$a='apple';

echo "i have an $ab"; //PHP解析器解析变量$ab为空

echo 'i hava an $a'; //单引号内不解析

echo "i have an ${a} b"; //花括号限制了变量名的边界

复杂语法即花括号语法，可以使用复杂的表达式：

<?php

$a='fantastic';

echo "this is $a bcd\n"; #有效

echo "this is ${a} bcd\n"; #有效

echo "this is {$a} bcd\n"; #有效

$fantastic='haha';

echo "this is ${$a}"; //可变变量

$arr=['cc' => [1,2,3]];

echo "there is an item of array in this string {$arr['cc'][1]}"; #有效，字符串key要用单引号括起来

在花括号内可以使用对象

note：

函数、方法、静态类变量和类常量只有在PHP5以后才可以再{$}中使用，而且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，只使用单一花括号{}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。

<?php

class beers{

const softdrink='rootbeer';

public $ale='ipa';

}

$rootbeer='A & W';

$ipa='alexander keith\'s';

$cc=new beers();

echo "i'd like an {${beers::softdrink}} \n"; //可变变量

echo "i'd like an {$cc->ale}";

字符串类似数组可以用索引访问（从0开始），也可以将下标放在花括号中

<?php

echo PHP\_EOL;

$str="Html\mannual\_files\_all\enhancingProgram\PHP";

echo $str[10] . PHP\_EOL;

echo $str{10} . PHP\_EOL; #也可以是使用花括号来访问

echo $str;

下标还可以修改字符串的值：

<?php

$str="afafafa";

echo $str . PHP\_EOL;

$str[10]='eDD'; #可以使用下标修改值，但是只会利用赋值字符串中的第一个字符

echo $str;

字符串使用.操作符连接；字符串加密解密函数mcrypt和mhash

一个值可以在其前面加上强制转换(string)来转换为字符串，或者使用strval()来转换为字符串；而且在表达式需要一个字符串时会自动转换。Settype也可以转换原字符串的类型。

数组强制转换为字符串值总为Array,~~对象总是被转换为Object~~，资源resource总是被转换成Resource id#1,null总是被转为空字符串

<?php

$num=123;

$str=strval($num);

var\_dump($str);

var\_dump(strval(false)); #输出空字符串''

var\_dump((string)true); #输出1

<?php

class Test{

}

$arr=[1,2,3];

echo (new Test()); //直接报错error

echo $arr;

大部分的PHP值可以转变为string来永久保存，这被称作串行化，使用函数serialize()

<?php

$arr=[1,2,3];

$arr\_ser=serialize($arr);

echo $arr\_ser . PHP\_EOL;

$arr\_unser=unserialize($arr\_ser);

var\_dump($arr\_unser);

字符串转换为数值：

如果该字符串没有包含. E 或E并且数字值在整型的范围内(由PHP\_INT\_MAX定义)，该字符串将被当做integer来取值，否则被当做float来取值；字符串的开始部分决定了它的值，如果以合法的数值（正负号 数字 小数点 指数部分的e或E）开头则使用该值，否则全部转换为0。

<?php

$a='123';

echo $a+1 . PHP\_EOL; //输出124

$b='1e+3';

echo $b+1 . PHP\_EOL; #输出1001

$c='+12e-2';

echo $c+1 . PHP\_EOL; #输出1.12

$d='12.2e+2';

echo $d+1 . PHP\_EOL; #输出1221

$f='a12';

echo $f+1 . PHP\_EOL; #输出1

PHP中使用函数ord()和chr()实现ASCII和字符之间的转换：

<?php

$a='a';

echo ord($a) . PHP\_EOL;

echo chr(90);

PHP中的字符串的实现方式是由字节组成的数组在加上一个整数指明缓冲区长度，并无如何将字节转换为字符的信息，因此PHP没有单独的byte类型，它由字符串来代替了。

PHP会被按照该脚本文件相同的编码方式来编码，除了激活zend multibyte。

某些字符串操作函数会假定字符串的编码方式，因此当字符串编码不同时，使用这些函数会有异常值出现：

<?php

$str='á';

echo strtoupper($str); //输出 谩

### 数组

PHP中的数组实际上是一个有序映射。可以将它当做真正的数组、列表、散列表、字典、集合、栈、队列等。而且树形结构和多维数组也是可以的。

自PHP5.4起，可以使用短数组定义语法，用[]代替array()

<?php

$arr1=array(1,2,3,'a'=>'a');

$arr2=array(1,2,3,);

$arr3=[1,2,3,'a'=>'a']; #PHP5.4以后才可以使用这种方法

var\_dump($arr1);

var\_dump($arr2);

var\_dump($arr3);

key是integer或者字符串，value可以是任何类型（包括对象，资源等）

数组和对象不能作为键名

数组中包含多个同键名数据保留最后一个：

<?php

$arr=['a'=>'a','b'=>'b','a'=>'aa'];

var\_dump($arr);

输出：

array(2) {

["a"]=>

string(2) "aa"

["b"]=>

string(1) "b"

}

数组键强制转换

<?php

$arr=[1=>'a',1.5=>'b',true=>'c','1'=>'d']; //数组键的强制转换

var\_dump($arr);

输出：

array(1) {

[1]=>

string(1) "d"

}

Key是可选项，如果为指定PHP会自动使用之前用过的integer键名+1来作为新的键名：

<?php

$arr=[1,2,3,100=>'a',12]; //12的索引时101

var\_dump($arr);

数组单元可以使用索引来访访问，方括号与花括号的效果相同：

<?php

$arr=[1,2,3,4,5];

echo $arr[1];

echo $arr{3};

自PHP5.4之后，可以使用数组间接引用函数或方法调用的结果，之前只能用临时变量。

<?php

function get(){

return [1,2,3,4,'a'=>'a'];

}

echo get()['a'];

访问未定义的键值产生E\_NOTICE级别的错误信息，并且结果为null。

<?php

$arr=[1,2,3,'a','b'];

var\_dump($arr['c']); //输出错误并返回NULL

删除某键值对使用unset函数

<?php

$arr=[1,2,3];

$arr[]=3; //在末尾增加一个3

var\_dump($arr);

$arr[12]=12;

var\_dump($arr);

$arr[]=13; //这时的索引变为13

var\_dump($arr);

unset($arr[3]); //删除某键值对，但不会重建索引

var\_dump($arr);

unset函数删除某个键值对但不会重建索引：

<?php

$arr=[1,2,3,4];

var\_dump($arr);

unset($arr[3]); //索引不会重建

$arr[]=5; //索引为4

var\_dump($arr);

最后一步输出：

array(4) {

[0]=>

int(1)

[1]=>

int(2)

[2]=>

int(3)

[4]=>

int(5)

}

object对象

转换为对象：对象转换为对象不会发生任何变化；其他任何类型的值被（强制）转换为对象会创建一个内置类stdClass的实例；如果该值为null则新的实例为空；数组转换成对象将是键名称为属性名，并具有相对应的值，对于任何其他的值，这个新实例的名为scalar的成员变量将包含该值。

<?php

$str="abc";

var\_dump((object)$str);/\*输出：

class stdClass#1 (1) {

public $scalar =>

string(3) "abc"

}

\*/

$num = 12;

var\_dump((object)$num);/\*

class stdClass#1 (1) {

public $scalar =>

int(12)

}

\*/

$arr=[1,2,4,'a','b','c','d'=>'dd'];

var\_dump((object)$arr);/\*

class stdClass#1 (7) {

public ${0} =>

int(1)

public ${1} =>

int(2)

public ${2} =>

int(4)

public ${3} =>

string(1) "a"

public ${4} =>

string(1) "b"

public ${5} =>

string(1) "c"

public $d =>

string(2) "dd"

}\*/

<?php

$n=null;

$nobj=(object)$n; #一个空stdClass对象，也可以new stdClass

$nobj->cc=12;

var\_dump($nobj);

<?php

$cc="cc";

$ccObj=(object)$cc;

echo $ccObj->scalar;

对象中返回对象:

<?php

class Foo{

public $a='a';

public $b='b';

public function getA(){

return $this->a;

}

public function getB(){

return $this->b;

}

public function setA($a){

$this->a=$a;

return $this; #返回此对象

}

}

echo (new Foo)

->setA(12)

->getA();

resource资源类型

资源类型是一种特殊的变量，保存了到的外部资源的引用。资源是通过专门的函数来建立和使用的。

一般资源类型变量保存有打开文件、数据库连接、图形画布等特殊句柄，因此将其他类型转换为资源类型没有意义。

PHP会自动释放不再被引用的资源，因此很少手动释放资源，除了持久数据库连接。

<?php

echo get\_resource\_type("cc"); //这会出错，参数需要是资源类型

NULL类型

它的唯一可能值为NULL，该值不区分大小写，以下三种情况下变量为null:

1 、被赋值为null

1. 尚未被赋值的变量
2. 被unset处理的变量

<?php

$a=12;

$b;

var\_dump(is\_null($a)); //false

var\_dump(is\_null($b)); //true

$arr=[1,2,3,'a'=>'aa'];

unset($arr['a']); //键为a的元素被删除

var\_dump($arr);

转换为null: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，仅仅是返回null值而已。

<?php

$a=12;

$b=(unset)$a; #$a的值并为改变，仅仅是返回null值给$b,除非unset($a)

var\_dump($a);

var\_dump($b);

<?php

class Foo{

protected $bar;

public function \_\_construct(){

$this->bar=null;

var\_dump($this->bar);

unset($this->bar);

var\_dump($this->bar);

}

public function \_\_get($var){ #魔术方法

echo "GET " . $var;

}

}

new Foo;

只有不在引用的内存中的值才会被垃圾回收：

<?php

$a='ab';

$b= &$a;

unset($a); # ab还被$b引用

echo $b;

callback 回调类型

自PHP 5.4起可以使用callable类型指定回调类型callback;回调类型不止可以是简单函数，还可以是对象的方法，包括静态方法。

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（除了语言结构外）；

<?php

function my\_callback\_function(){

echo 'this is a my\_callback\_function';

}

call\_user\_func('my\_callback\_function'); #输出 this is a my\_callback\_function

function cc($cc){

$cc();

}

cc('my\_callback\_function'); #输出 this is a my\_callback\_function

一个已实例化的对象的方法作为数组传递，下表0包含该对象，下表1包含方法名。

静态类方法可以不经实例化该类的对象而传递，只要在下表为0中包含类名而不是对象，自PHP 5.2.3起可以这样传递静态方法：className::methodName

<?php

class MyClass{

static function mycallbackmethod(){

echo "myclass-->mycallbackmethod is used";

}

}

call\_user\_func(['MyClass','mycallbackmethod']); #输出 myclass-->mycallbackmethod is used

$myclass=new MyClass;

call\_user\_func(array($myclass,'mycallbackmethod')); #输出 myclass-->mycallbackmethod is used,

call\_user\_func('MyClass::mycallbackmethod'); #输出 myclass-->mycallbackmethod is used

<?php

class A{

public static function who(){

echo "A\n";

}

}

class B extends A{

public static function who(){

echo "B\n";

}

}

call\_user\_func(['B','parent::who']); //输出 A

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，自PHP 5.3.0起可以传递closure给毁掉函数。

<?php

$myFun=create\_function('$a,$b','return $a+$b;');

var\_dump($myFun(1,3)); //输出4

<?php

function myFun($cc){

echo 'ac';

}

myFun('cc',12); //这样写完全没有什么错误

类型转换的判别

PHP在变量定义中不需要明确的类型定义，变量的类型时根据使用该变量的上下文所决定。

自动转换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。

允许的强制转换有：

* (int), (integer) - 转换为整形 [integer](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.integer.html)
* (bool), (boolean) - 转换为布尔类型 [boolean](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.boolean.html)
* (float), (double), (real) - 转换为浮点型 [float](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.float.html)
* (string) - 转换为字符串 [string](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.string.html)
* (array) - 转换为数组 [array](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.array.html)
* (object) - 转换为对象 [object](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.object.html)
* (unset) - 转换为 [NULL](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/language.types.null.html) (PHP 5) （仅仅是返回一个null，原操作数并没有改变）
* <?php
* $a=0;
* var\_dump((bool)$a);
* var\_dump((boolean)$a); #bool和boolean都是可以的

(binary) 转换和 b 前缀转换支持为 PHP 5.2.1 新增。

<?php

$str='start up : igbinary';

$strNew=b'start up : igbinary'; #这样也可以在开始时转换为binary

$b\_str= (binary)$str;

echo gettype($b\_str); #输出string

内置标准类实例化不区分大小写，而且可以有数字属性：

<?php

$num='12';

var\_dump(is\_numeric(12)); #判断是否为数字类型或可转换为数字类型

$s=new stdclass; #不区分大小写

$s->{"2"}=12; #数字属性

var\_dump($s->{"2"}); #访问数字属性

## 变量

PHP的变量名是区分大小写的，美元符号$后面跟变量名

只有有名字的变量才可以引用赋值

预定义变量 全局变量 超全局变量

<?php

var\_dump($\_ENV);

var\_dump($\_SERVER['CLASSPATH']);

可变变量-就是变量名可变

<?php

$a="b";

$$a=12;

echo $b; #只有$a=b时才会有变量$b

#### 变量的作用域：

在用户自定义函数中一个局部函数范围将被引入，任何用在函数内部的变量将按缺省的情况被限制在就局部函数范围内，除非用关键字global声明：

<?php

$a=12;

function a(){

echo $a;

}

function b(){

global $a;

$a=10;

echo $a;

}

a(); //未找到变量值

b(); //输出10

echo $a; //输出10

在函数范围内访问全局变量的另一种方法是使用超全局关联数组$GLOBALS

<?php

var\_dump($GLOBALS);

变量范围的另一个重要特征是静态变量，它仅仅存在局部函数域，但当程序执行离开此作用域时其值并不消失。

<?php

$a=12;

function myFun(){

static $a=0; 仅仅在第一次调用函数时初始化

$a++;

echo $a;

}

myFun(); //1

myFun(); //2

myFun(); //3

myFun(); //4

echo $a; //12

变量赋值为函数，也可以将函数作为返回值

<?php

$a=function(){

echo 'a';

}; #最后的分号必须

$a();

<?php

function c(){

global $a; //相当于在全局注册了变量$a

$a=1;

}

c();

echo $a; //输出1

由类名实例化类：

<?php

class test1{}

class test2{}

class test3{}

$a='test1';

var\_dump(new $a()); //这也可以

### 可变变量

就是变量的名字可以改变

<?php

$a='hello';

$$a='world';

echo "$a $hello"; //这时两个变量都被定义了

$a='change';

echo $hello;

当将可变变量用于数组时，必须使用花括号明晰边界，也可以使用可变变量操作类的属性，当遇到数组时也要使用花括号{}来明晰边界。

来自PHP之外的变量

$\_GET $\_POST $\_COOKIE $\_REQUST

#### 常量

可以用define函数来定义常量，自PHP 5.3.0以后，可以使用const关键字在类定义之外定义产量，一个常量一旦被定义就不允许在改变或取消定义。

<?php

const cc=12;

var\_dump(cc); //常量不用使用$

常量只能使用标量数据（boolean integer float string），也可以定义resource常量但应尽量避免。

<?php

const cc=12;

var\_dump(cc); //常量不用使用$

$a='aa';

var\_dump(get\_defined\_constants()); //获取所有常量

<?php

// const cc=12;

var\_dump(defined(cc)); //返回false，应该输入字符串名称

define("cc",12);

var\_dump(defined(cc)); //返回false，应该输入字符串名称

var\_dump(cc); //常量不用使用$

$a='aa';

// var\_dump(get\_defined\_constants()); //获取所有常量

变量和常量的不同：

* 常量前面没有美元符号（$）；
* 常量只能用 [define()](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/function.define.html) 函数定义，而不能通过赋值语句；
* 常量可以不用理会变量的作用域而在任何地方定义和访问；
* 常量一旦定义就不能被重新定义或者取消定义；
* 常量的值只能是标量。

使用 const 关键字定义常量必须处于最顶端的作用区域，因为用此方法是在编译时定义的。这就意味着不能在函数内，循环内以及 if 语句之内用 const 来定义常量。

#### 魔术常量

\_\_LINE\_\_：文件中当前的行号

\_\_FILE\_\_：文件的完整路径和名称。

\_\_DIE\_\_：文件所在的目录，等价于dirname(\_\_FILE\_\_)

\_\_FUNCTION\_\_：当前函数的名称

\_\_CLASS\_\_：当前类的名称

\_\_TRANT\_\_：trait的名字

\_\_METHOD\_\_：类的方法的名称

\_\_NAMESPACE\_\_\_：命名空间的名称，区分大小写

<?php

namespace ab;

var\_dump(\_\_FILE\_\_); //输出：E:\deleting\enhancingProgram\PHP\codes\yuyan\_cankao\69.php

var\_dump(\_\_DIR\_\_); //输出：E:\deleting\enhancingProgram\PHP\codes\yuyan\_cankao

function cc(){

echo \_\_FUNCTION\_\_;

}

cc(); //输出cc

class ab{

function \_\_construct(){

echo \_\_CLASS\_\_; //输出ab

echo \_\_METHOD\_\_; //输出 ab::construct

}

}

new ab;

echo \_\_NAMESPACE\_\_; //输出ab

get\_class()返回对象的类名

get\_object\_vars返回由对象中定义的属性组成的关联数组

file\_exists判断文件是否存在

function\_exists判断方法是否存在

<?php

function cc(){

}

$cc='cc';

var\_dump(function\_exists($cc)); //true

const ab='ab';

var\_dump(defined('ab')); //传入的是常量名

表达式

表达式就是任何有值的东西，php是一种面向表达式的语言。

PHP支持全等运算符===，要求值和类型都相等才是true，非全等!==

相等判断时并不一定由右至左运算：

<?php

function a(){

echo 'a';

}

function b(){

echo 'b';

}

var\_dump(a()==b()); //输出abbool(true),相等判断时并不一定由右至左

运算符

PHP允许这样的赋值和判断：

<?php

$a=12;

if(!$a=10){ //先赋值在取反判断

echo $a;

}else{

echo $a;

}

PHP中的赋值运算大都是传值赋值，除了对象object(除非使用了clone关键字)

<?php

class Test{

public $a;

}

$a=new Test;

$b=$a;

$a->a=12;

echo $b->a . PHP\_EOL; #12

$b->a=10;

echo $a->a . PHP\_EOL; #10

new 运算符自动返回一个引用，不能与&重用。

位运算符：

& | ^ ~ <<(每一次都表示乘以2) >>（每一次都表示除以2）

<?php

$a=2;

echo $a<<10 . PHP\_EOL; #2048

echo ~$a . PHP\_EOL; #-3

echo $a & 3 . PHP\_EOL; #2

echo $a | 5 . PHP\_EOL; #7

echo $a^1 . PHP\_EOL; #3

位移在 PHP 中是数学运算。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。

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。

比较运算符：

=== 值和类型都相等返回true，对应的!==

!= 或者 <>

当比较数字和数字字符串时会将字符串转换为数字

不应比较两个浮点数的大小，可能出错

##### 错误控制运算符

PHP支持一个错误控制运算符@，将它放在一个PHP表达式之前，任何的错误信息都将被忽略。

如果用set\_error\_handler设定了自定义错误处理函数，即使使用了@，自定义函数仍然会被调用，但是此处理函数可以调用error\_reporting，而该函数在出错语句前有@时返回0；

@运算符只对表达式有效，可以将它发在变量、函数和include调用、常量等之前。不能把它放在函数或者类的定义之前，也不能用于条件结构如if和foreach之前。

执行运算符：反引号--``

PHP将尝试将反引号中的内容作为外壳命令来执行，并将其输出信息返回。

<?php

$output=` dir `; //可执行shell命令

var\_dump($output);

此效果等同于shell\_exec()，但是当激活了安全模式或者关闭了shell\_exec()时是无效的。而且反引号不能再双引号字符串中使用。

<?php

$output=` dir `; //可执行shell命令

var\_dump($output);

$cc=shell\_exec("dir");

var\_dump($cc);

<?php

$cc=` php 75.php`; #执行75.php，执行结果复制到$cc

var\_dump($cc);

var\_dump(gettype($cc)); #string

递增与递减运算：

递增和递减运算不影响布尔值。null递减还是null，递增的话为1。

<?php

$t=true;

$f=false;

$t++; //布尔值递增还是true

var\_dump($t);

$f--;

var\_dump($f); //布尔递减还是false

$c=null;

var\_dump(--$c); #null

var\_dump(++$c); #1

处理字符串时沿袭了perl的习惯，$a=’Z’;$a++得到的结果为AA，纯字母字符串只能递增不能递减，其他字符串递增和递减不会变化。

逻辑运算符：

and 和&& or 和 || not 和! ，字母的与符号的区别就是运算符的优先级不同，其他一样，都会有短路操作。 xor

<?php

$a=false;

$b=true;

var\_dump($a || $b); #true

var\_dump($a or $b); #true

$a=1;

var\_dump(!$a); //输出false

逻辑运算符并不会像js一样赋值

<?php

$a=false;

$cc=(false or "jack");

var\_dump($cc); //true，并不会得到jack

但是利用短路的规则进行赋值：

<?php

$a=12;

$b="cc";

$a>10 && $b=12;

echo $b; //输出12

字符串运算符：

字符串连接运算符.和连接赋值运算符.=

数组运算符：

+运算符把右边的数组元素附加到左边数组的后面，两个数组都有的键名只用左边数组的，右边的会别忽略。

|  |  |  |
| --- | --- | --- |
| $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** 。 |

<?php

$arr1=[1,2,3,4];

$arr2=['a','b','c','d','e'];

var\_dump($arr1+$arr2); //数组联合，如果两个数组具有相同的键则取第一个数组的键

向数组添加元素：

<?php

$arr=[];

$arr[]=1; //将1添加到为数组的最后一个元素，相当于array\_push($arr,1)

array\_push($arr,1);

var\_dump($arr);

类型控制运算符：

instanceof用于确定一个php变量是否属于某一类（或该类的子类）或接口的实例

<?php

class Test1{

}

class Test2 extends Test1{

}

$t=new Test2;

var\_dump($t instanceof Test1); #输出true

var\_dump($t instanceof Test2); #输出true

<?php

interface Test1{

}

class Test2 implements Test1{

}

$t=new Test2;

var\_dump($t instanceof Test1); #接口实现类的实例也会输出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（不能直接使用字符串）：

<?php

class Test1{

}

$t=new Test1;

$t2=new Test1;

var\_dump($t instanceof Test1); #true

$cName="Test1";

var\_dump($t instanceof $cName); #true，注意不要直接使用字符串，会报错

var\_dump($t instanceof $t2); #true

<?php

namespace test;

class Test1{

}

$t=new Test1;

var\_dump(get\_class($t)); //test\Test1 输出包括命名空间

输出的类名带有命名空间，如果没有就是全局空间

<?php

namespace test;

class Test1{

}

$t=new Test1;

$cc=new \stdClass; #必须指明命名空间，否则只会在当前空间查找

var\_dump(get\_class($t)); //test\Test1

var\_dump(get\_class($cc)); //stdClass 表明是全局命名空间

### 流程控制

一条语句以分号结束，还可以用花括号将一组语句封装成一个语句组，语句组可以被当做一行语句（类似c#中）

<?php

$a='a';

$b='b';

{

$c=1;

$d=2;

}

var\_dump($c); //1

var\_dump($d); //2

if语句的另一种写法（多用在嵌套在html中）

<?php

$a=true;

if (!$a):

echo 1;

else:

echo 2;

endif;

使用：表示法来执行if else等语句时不可以使用else if，要写成一个单词

<?php

$a=true;

if($a):

echo 1;

else{ //格式一致，需要使用：格式

echo 2;

}

流程控制的替代语法：

替代语法就是将做花括号变为冒号:，结尾用endif 、 endwhile 、 endswitch 、 endfor 、endforeach

for语句的条件项可以省略

<?php

$i=1;

for(;$i<10;){

echo $i;

$i++;

}

可以使用goto语句跳出多层循环，在循环条件中写多个表达式：

<?php

for($a=1,$b=2;;):

{

echo $a;

$a++;

if($a>10){

break;

}

}

endfor;

foreach仅能够用于数组和对象，有两种用法：

foreach($var as $k => $v){}

foreach($var as $v){}

foreach的遍历依赖于数组内部的指针，不要试图在循环内部修改数组的指针

foreach不会改变遍历的变量的值（除非是引用类型或者使用引用赋值&），可以使用for循环

引用赋值：foreach($arr as $k => &$v){，当使用引用赋值的时候需要在最后销毁赋值的变量（因为它仍然存在，即unset($v)）

foreach中不能使用@来抑制错误消息的输出

<?php

$arr=[1,2,3,4,'a'=>'aa','b'=>'bb'];

foreach($arr as $k => $v){

if(is\_int($v)){

$v\*=2;

}

if(is\_string($v)){

$v .= $v;

}

}

var\_dump($arr); //在foreach中对数组的改变，原数组不改变

<?php

$arr=[1,2,3];

for($i=0;$i<count($arr);$i++){

$arr[$i]\*=2;

}

var\_dump($arr); //原数组发生了改变

在foreach中可以使用list对多维数组的数组元素进行解包：

<?php

$arr=[

[1,2,'a'],

[3,4,'b'],

[5,6,'c']

];

foreach($arr as list($a,$b)){ //a和b临时解析为数组元素中的元素，可以比数组元素的个数少但是不能多

echo '$a: ' . $a . ' | ' . '$b: ' . $b . '<br />';

}

函数中定义的全局变量会存在于全局，即使函数执行完全局变量也不会消失：

<?php

function a(){

global $cc; //这里的变量定义，如果全局没有此变量会创建

$cc=12;

}

a();

echo $cc; //12

each返回一个数组：

<?php

function a(){

global $cc; //这里的变量定义，如果全局没有此变量会创建

$cc=12;

}

a();

echo $cc; //12

$arr=[1,2,3];

print\_r(each($arr)); #输出：

Array

(

[1] => 1

[value] => 1

[0] => 0

[key] => 0

)

最好在每一次foreach后面使用unset方法销毁掉迭代变量，如unset($k,$v)

break可接受可选的数字参数来决定跳出几层循环

<?php

$i=0;

for(;;){

for(;;){

for(;;){

echo "1";

if($i>10){

break 3; #跳出3层循环

}

$i++;

}

}

}

continue接受一个可选的数字参数来决定跳过几重循环到循环结尾，默认值是1>

switch/case做的是松散比较，不要忘了break语句，default的位置不固定

<?php

$a=1;

switch($a){

default:

echo 'default';

break;

case 1:

echo '1'; //即使default写在了前面也会执行先case，直到所有case后才执行default

}

Declare结构用来设定一段代码的执行指令，语法和其他流程控制语句相似：

Declare (directive)

Statement

Directive不服你设定declare代码段的行为，目前只认识两个指令：ticks和encoding（PHP5.3.0新增）

Declare结构可用于全局范围，影响到后面的所有代码（如果包含declare结构的子文件被包含，父文件不受影响）

Ticks：是一个在declare代码段中解释器每执行N条可计时的低级语句（通常条件表达和参数表达式都不可计时）就会发生的事件；在每个tick中出现的事件是由register\_tick\_function()来指定的。

可以用来对某个过程计时，如数据连接、网络连接等

declare (ticks=1);

function tick\_handler(){

echo "tick\_handler() called\n";

}

register\_tick\_function('tick\_handler');

$a=1;

if($a>0){ //左花括号不算计时语句

$a+=2;

echo $a;

} //这也算计时语句

输出：

tick\_handler() called

tick\_handler() called

tick\_handler() called

3tick\_handler() called

tick\_handler() called

Encoding可以对每段脚本指定其编码方式：

<?php

declare(encoding="UTF-8");

$a=1;

Return结构：

如果在全局范围内调用，则当前脚本中止运行；如果是当前脚本是被包含的，则return的值会被当做require、include调用的返回值。

Return是结构不是函数，不需要加花括号；

如果函数没有return则默认返回null

<?php

$c=require\_once('108.php');

var\_dump($c);

如果被包含的文件定义了函数，那么也可以在包含文件中调用（在当前文件中不带命名空间的函数调用都默认当前的命名空间）；当存在命名空间时需要额外的处理：

<?php

namespace cc;

require\_once '109.php';

\a\cc();

Include和require都可以包含外部文件，都是语言结构因此不需要花括号，两者几乎完全一样，只是当出现错误时include产生警告但不会中止脚本，而require会中止脚本执行。

Include和require包含文件时，对文件的查找有一个规则；另外还可以通过http包含远程文件。

如果被包含文件不使用return返回值的话，包含成功返回1，失败include会发出警告，require会出错。

包含文件时可以使用chdir改变当前的路径，引入文件后再回到之前的路径。

Include\_once和require\_once和不带once的语句类似，唯一的区别是会判断文件是否被包含过，如果已经被包含则不再包含（避免函数重定义，变量重新赋值）。

命名空间被包含时不会作用于变量，即变量仍然可以直接写就如同在当前文件中定义过一样，而函数则不同，需要命名空间前缀：

<?php

namespace b;

require '111.php';

echo $cc;

\a\d();

goto 操作符可以用来跳转到程序中的另一位置，goto语句后面不能是变量，需要时名字标记。该目标位置可以用目标名称加上冒号来标记，而跳转指令是 goto 之后接上目标位置的标记。PHP 中的 goto 有一定限制，目标位置只能位于同一个文件和作用域，也就是说无法跳出一个函数或类方法，也无法跳入到另一个函数。也无法跳入到任何循环或者 switch 结构中。可以跳出循环或者 switch，通常的用法是用 goto 代替多层的 break。PHP5.3.0以上版本有效。

<?php

$a=1;

goto a;

echo 'aa';

$b=12;

a:

echo "cc"; //只会输出cc，不会输出aa

<?php

$a=12;

echo $a;

goto a;

echo 'a';

a:{

$b='b';

echo $b;

goto b;

echo 'bb';

b:{

echo 'c';

goto c;

echo 'cc';

}

c:

goto d; //只能是a b c 但不能是a外面的

d:{

}

}

## 函数

任何有效的PHP代码都可能出现在函数内部，包括其他函数或者类的定义。

函数无非在调用之前被定义，但调用时必须保证函数的定义被执行到。

<?php

function foo(){

echo "foo";

function bar(){

echo "bar";

}

}

foo(); //只有调用了foo函数之后，bar函数才被定义，从而才能调用

bar();

函数中包含类：

<?php

function test(){

class A{

public $a;

public $b;

public function fa(){

echo \_\_CLASS\_\_ . PHP\_EOL;

echo \_\_FUNCTION\_\_ . PHP\_EOL;

}

}

}

// $ca=new A; #这句执行会报错，找不到类A的定义

test(); #执行完这句才可以实例化类A

$ca=new A;

$ca->a=12;

var\_dump($ca->a);

var\_dump($ca->b);

$ca->fa();

PHP中的所有函数和类都有全局作用域，可以定义在一个函数之内而在之外调用，也可以在函数内调用所在函数外定义的函数（要带命名空间前缀，否则默认在同一命名空间查找）

PHP中有个限制，函数嵌套不能超过100层（数字可配置），可以将函数定义赋值给数组的元素，函数名是大小写无关的，但是尽量要规范命名。

函数的参数默认是按值传递的（因而即使在函数内部改变了参数的值但是并不会改变函数外部的值），如果想要函数的参数引用传递则在定义函数时在参数前加上符号&

<?php

function noRef($a){

$a=1;

echo $a . PHP\_EOL; #输出1

}

function withRef(&$a){ //定义的时候引用传递参数

$a=2;

echo $a . PHP\_EOL; #输出2

}

$a=12;

noRef($a);

echo $a . PHP\_EOL; #输出12

withRef($a); //改变了外面的数值

echo $a; #输出2

参数可以有默认值，任何默认参数必须放在非默认参数的右侧

函数定义时甚至可以不指定参数，而在调用的时候传递进任意数量的参数，但是这些传递的参数不能使用参数名来使用。可以使用func\_get\_arg来使用第n个参数

<?php

function a($a,$b,$c){

var\_dump(func\_num\_args()); //返回时传递参数的个数

var\_dump(func\_get\_args()); //返回所有参数组成的索引数组，下表从0开始

var\_dump(func\_get\_arg(2)); //输出下表为2的参数

var\_dump($a);

}

a(1,2,3,4,5,6,7);

可以指定函数参数的类型（但是这个类型必须是自定义类型），也可以指定函数返回值的类型（需要PHP7）

<?php

function a(int $c){ //为参数指定类型，这个int类型需要自己定义，与内置的类型不是一回事

echo $c;

}

a("12"); //fatal error

function b(bool $c):string{ //这句需要PHP 7

var\_dump($c);

return "";

}

<?php

class string{

function \_\_toString(){

return \_\_CLASS\_\_;

}

}

function a(string $c){ //为参数指定类型,这个类型需要是自定义类型

echo $c;

}

$a=new string;

a($a); //fatal error

// function b(bool $c):string{ //这句需要PHP 7

// var\_dump($c);

// return "";

// }

函数返回值

返回语句立即中止函数的裕兴，并且将控制权交给调用函数的代码行。如果省略了return语句则返回null

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 & a($a){

$a=10;

return $a;

}

$c=12;

$b=& a($c);

var\_dump($b);

var\_dump($c);

可变函数：

这意味着一个变量名后面有圆括号，PHP将寻找与变量的值同名的函数，并且尝试执行它。

<?php

function foo(){

echo "In foo()" . PHP\_EOL;

}

function bar($arg=""){

echo $arg;

}

$func='foo';

$func(); //这句执行foo函数

$func="bar";

$func("In bar()" . PHP\_EOL);

类似的也有可变类（可能：PHP遇到new关键字后就会找与后面的参数名字相同的类，然后实例化出一个对象）：

<?php

class A{

public function \_\_construct(){

var\_dump(\_\_CLASS\_\_);

var\_dump(\_\_FUNCTION\_\_);

}

}

class B{

public function \_\_construct(){

var\_dump(\_\_CLASS\_\_);

var\_dump(\_\_FUNCTION\_\_);

}

}

$className="A"; //可变类

$a=new $className;

var\_dump($a);

$className="B";

var\_dump(new $className);

内置函数：

扩展的函数需要和特定的PHP扩展模块一起编译，否在会找不到函数。

匿名函数：

匿名函数也叫闭包函数，最常用作回调函数的参数值，也可以用作变量来使用。

闭包函数也可以作为变量的值来使用，PHP会自动把此种表达式转换为内置类Closure的对象的实例（即被赋值为匿名函数的变量会自动转换为closure的实例）。把一个closure对象赋值给一个变量的方式与普通变量的赋值的语法是一样的。

<?php

$a=function(){

echo "a";

};

$b=function(){

echo "b";

};

var\_dump($a);

var\_dump($b);

输出：

class Closure#1 (0) {

}

class Closure#2 (0) {

}

Closure 对象也会从父作用域中继承类属性。这些变量都必须在函数或类的头部声明。从父作用域中继承变量与使用全局变量是不同的。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。而 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）。

constant(string)返回一个常量的值。

call\_user\_func调用用户自定义函数，可选出传递参数，一个一个给出

call\_user\_func\_array调用用户自定义函数，参数以数组的形式传递

### 类与对象

PHP对待对象的方式与引用和句柄相同，即每个变量都持有对对象的引用，而不是整个对象的拷贝。

当一个方法类的内部调用时可以使用伪变量$this;$this是一个到主叫对象的引用

要创建一个类的实例必须使用new关键字，如果new之后跟着的是一个包含类名的字符串，则该类的一个实例被创建，如果该类属于某个命名空间，则必须使用完整的名称。

<?php

class A{

}

$clsName="A";

$a=new $clsName; //这样也能创建一个类

var\_dump($a);

$ab=new A;

var\_dump($ab); //A的第二个示例

属于某个命名空间的类实例化要带完整的名称：

<?php

namespace test;

class A{

}

$clsName="\\test\A";

$a=new $clsName; //必须使用完整而名称，带命名空间

var\_dump($a);

使用new self和new parent创建 自身的对象和其他类的对象

<?php

class A{

}

class B extends A{

public static $b;

public function ba(){

$this->b=new self;

$pa=new parent;

var\_dump($this->b);

var\_dump($pa);

}

}

$bo=new B;

$bo->ba();

对象赋值时是引用赋值，当个函数传递一个对象时也是引用传递即在函数内部对对象的改变会改变原对象的表现；可以使用克隆方法给已创建的实例复制一份完全一样的实例，这时两个实例访问的是不同的。

<?php

class test{

public $a;

public $b;

}

$t1=new test;

$t1->a=12;

$t2=$t1;

var\_dump($t2->a);

$t2->b=21;

var\_dump($t1->b);

$t3=clone($t1); //$t3是test的第二个实例

var\_dump($t3);

$t3->a=123;

var\_dump($t1->a); //没有改变，还是12

当使用var\_dump输出对象的时候，会打印出实例是类的第几个实例，如下：

class test#2 (2) { #表示是第二个实例

public $a =>

int(12)

public $b =>

int(21)

}

给对象不存在的属性赋值会为对象创建此属性：

<?php

error\_reporting(0); //关闭所有错误报告

class test{

public $a;

public $b;

}

$t1=new test;

$t1->c="cc"; //对象多了一个属性cc

var\_dump($t1);

var\_dump($t1->dd); //notice：输出null

创建类的实例的三个方法：new 类名；new 类名字符串变量;new 该类的实例

<?php

class A {

public $a;

}

class B{

}

$a1=new A;

$a1->a=12;

$a2=new $a1; //这时创建了A的第二个实例,等价于new A

var\_dump($a2);

类的继承关键字是extends,PHP也不支持多继承，被继承的方法和属性可以通过同样的名字重新声明被覆盖，除了父类定义时使用了final关键字（只能修饰类和方法）。子类中可以通过parent::来访问被覆盖的方法或属性。

覆盖方法时参数必须保持一致，构造方法除外。

<?php

class a{

public $a1;

public $a2;

public function \_\_construct(){

$this->a1='a1';

$this->a2='a2';

}

public function canwrite(){

echo \_\_CLASS\_\_ . '/' . \_\_FUNCTION\_\_;

}

public final function nowrite(){

echo \_\_FUNCTION\_\_;

}

}

class b extends a{

public $a1;

public $a2;

public $b1;

public function \_\_construct($c){

$this->a1='b->a1';

$this->b1='b1';

}

public function canwrite(){

parent::nowrite();

parent::canwrite();

echo \_\_CLASS\_\_ . '/' . \_\_FUNCTION\_\_;

}

// public function nowrite(){ //不能重写final关键字修饰的方法

// echo \_\_FUNCTION\_\_;

// }

}

$bo=new b("");

var\_dump($bo->a1);

var\_dump($bo->a2);

$bo->canwrite();

Self关键字和parent关键字都要使用：：符号

使用className::class可以得到类的完全限定名称，它与\_\_CLASS\_\_一样，而且可以在类的外部使用：

<?php

namespace test;

class a{

}

var\_dump(a::class); //等同于在类的内部使用的\_\_CLASS\_\_

私有的构造方法可以将类做成单例，获得单例的方法需要是静态的。

<?php

namespace test;

class a{

private $a;

private function \_\_construct($var){

$this->a=$var;

}

public function get(){

return $this->a;

}

public static function getInstance($v){

return new self($v); #使用new a也可以

}

}

$obj=a::getInstance("aa");

var\_dump($obj);

var\_dump($obj->get());

类的变量成员叫做属性或者字段、特征。关键字修饰为public protected private,默认为public，可以初始化但必须是常量值。访问非静态属性使用对象操作符->，访问静态属性则使用::;如$tihs->属性名（不带$符号），而self::$静态属性名（带$符号）

类常量

在定义和使用类常量的时候不要使用$符号，接口中也可以定义常量。

自PHP 5.3.0起可以用一个变量来动态调用类（该变量的值不能为关键字），就像之前的字符串变量来实例化类

<?php

class a{

const CONSTANT='NOT CHANGED';

public $a='a';

}

var\_dump(a::CONSTANT);

$clasName="a";

var\_dump($clasName::CONSTANT); #这也可以

自动加载类

构造函数和析构函数：

如果子类中中定义了构造函数则不会隐式的调用其父类的构造函数，可以使用parent::\_\_construct来显示的调用父类的构造函数。如果子类没有构造函数则会从父类继承构造函数。

为了实现向后兼容性，PHP5在类中找不到\_\_construct函数并且也没有从父类继承一个的话，他就会寻找旧式的构造函数（即与类名相同的函数）

<?php

class a{

public function a(){

print \_\_CLASS\_\_;

print \_\_FUNCTION\_\_;

}

}

new a;

析构函数：

<?php

class a{

public function \_\_construct(){

echo "this is \_\_construct" . PHP\_EOL;

}

public function test(){

echo "this is test" . PHP\_EOL;

}

public function \_\_destruct(){

echo "this is destruct" . PHP\_EOL;

}

}

(new a)->test();

和构造函数一样，父类的析构函数不会显示的被引擎调用，要执行父类的析构函数，必须在子类中显示的调用parent::\_destruct(),此方法也和构造函数一样，如果子类没有定义析构函数则继承父类的。

<?php

class a{

public function \_\_construct(){

var\_dump("this is construct");

}

public function \_\_destruct(){

var\_dump("this is destruct");

}

}

class b extends a{ //没有定义自己的则继承自父类的

public function tt(){

var\_dump(\_\_FUNCTION\_\_ . 'is called');

}

}

$b=new b;

$b->tt();

exit("stop to execute" . PHP\_EOL); //先执行exit函数，再执行析构函数

访问控制（可见性）：

关键字：public protected private

public:可以在任何地方访问

protected:受保护的类成员可以被其自身以及其子类和父类访问；

private：私有类成员则只能被其定义所在的类访问。

类属性必须定义为public protected private之一，默认为public

方法的访问控制符同属性的。

<?php

class a{

public $a;

public $b;

protected $pa;

protected $pb;

private $ppa;

private $ppb;

}

class b extends a{

public function getApa(){

return (new a)->pa;

}

public function getAppa(){

return parent::ppa;

}

}

$a=new a;

$b=new b;

var\_dump($b); //貌似从类a继承了私有和保护属性但是不能访问

// var\_dump($a->ppa); //对象不能在外部访问私有属性ppa

var\_dump($b->getApa()); //可以访问父类的protected属性

// var\_dump($b->getAppa()); //父类私有属性不能再类外部访问

<?php

class a{

private $a;

public function setA($var){

$this->a=$var;

}

public function getA(){

return $this->a;

}

}

$a1=new a;

$a2=new a;

$a1->setA(12);

var\_dump($a1->getA()); //12

var\_dump($a2->getA()); //null

### 对象的继承

当扩展一个类，子类就会继承父类所有公有和受保护的方法，除非子类覆盖了父类的方法

<?php

class p{

public $a;

public $b;

public function pa(){

echo \_\_CLASS\_\_ . '/' . \_\_FUNCTION\_\_ . PHP\_EOL;

}

public function pb(){

echo \_\_CLASS\_\_ . '/' . \_\_FUNCTION\_\_ . PHP\_EOL;

}

}

class c extends p{

public $a;

public $c;

public function pa(){

echo "this is in child" . PHP\_EOL;

}

public function pc(){

echo "this is a method in child" . PHP\_EOL;

}

}

$c=new c;

$c->pa(); //覆盖了父类的方法

$c->pb();

$c->pc(); //子类自己的方法

范围解析操作符：

一对冒号：：，可以访问静态成员、类常量，还可以用于覆盖类中的属性或者方法。

自PHP5.3.0起，可以通过变量来引用类，该变量的值不能是关键字（self parent或者static）

self parent和static 这三个关键字时用于在类内部对其属性进行访问的。

<?php

class a{

const A=12;

}

$clasName="a";

var\_dump($clasName::A); //变量引用类来访问常量

当一个子类覆盖父类中的方法时，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。

static：：关键字可以访问静态方法、变量或常量。

const前面不能有访问控制符

### 静态关键字

Static可以代表当前类（随上下文变化，如继承中），所以可以在类中使用new static得到当前类的实例。

声明类属性或方法为静态，就可以不实例化类而直接访问（因为这些属性或方法是属于类的），静态属性不能通过一个类已实例化的对象通过->来访问,要用：：（但静态方法可以-类似于JAVA）

由于静态方法不需要通过对象就可以调用，所以伪变量$this在静态方法中不可用。

静态属性只能被初始化为文字或常量，不能使用表达式，所以可以把静态属性初始化为整数或数组，但不能初始化为另一个变量或函数返回值，也不能初始化为另一个对象。

静态变量引用的时候要带$符号，普通变量反而不用

在类内部，静态变量和方法可以使用static 或者self访问，在类外部用类名访问。

<?php

class test{

public $a;

public static $b;

public function \_\_construct($a,$b){

$this->a=$a;

static::$b=$b;

self::$b=$b;

}

public static function c($c){

self::$b=$c;

}

}

$a=new test(1,2);

var\_dump($a->a);

var\_dump(test::$b);

test::c(12);

var\_dump(test::$b); //12

$clasName="test";

var\_dump($clasName::$b); //可以使用变量来引用类

静态变量是属于类的，因此即使两个对象属于不同的实例，他们访问的静态变量是一样的，会相互影响：

<?php

class t{

public static $a;

}

$a=new t;

$b=new t;

var\_dump($b::$a);

$a::$a=12;

var\_dump($b::$a);

静态变量需要使用::来访问（在类的内部static或者self或者parent，在类的外部使用类名或对象或值为类名的变量），静态方法可以使用::或者->来访问（在类的内部self或者static或者parent,在类的外部使用类名或对象或值为类名的变量）

常量就相当于是不能更改的静态变量，也使用此规则。

Self关键字强制使用调用时的当前类，而static会随着类的不同而改变：

<?php

class a{

static protected $test="class a";

public function static\_test(){

echo static::$test; // Results class b

echo self::$test; // Results class a

}

}

class b extends a{

static protected $test="class b";

}

$obj = new b();

$obj->static\_test();

单例的创建及使用：

<?php  
  
abstract class Singleton {  
  
    protected static $\_instance = NULL;  
  
    /\*\*  
     \* Prevent direct object creation  
     \*/  
    final private function  \_\_construct() { }  
  
    /\*\*  
     \* Prevent object cloning  
     \*/  
    final private function  \_\_clone() { }  
  
    /\*\*  
     \* Returns new or existing Singleton instance  
     \* @return Singleton  
     \*/  
    final public static function getInstance(){  
        if(null !== static::$\_instance){  
            return static::$\_instance;  
        }  
        static::$\_instance = new static();  
        return static::$\_instance;  
    }  
      
}

class cc extends Singleton{ //抽象类不能实例化，需要其他类来实现

}

var\_dump(cc::getInstance()); //类cc,并且拥有抽象类定义的特性

$cc=new cc; //这句是不允许的，除非在cc类中重定义构造函数为公有方法，但如果父类构造函数为final则不能重写

静态属性不只能在类的不同对象中公用，也能够在子类和父类中公用（除非子类重写了父类的静态属性），即子类可以通过自身访问父类的静态属性

<?php

class a{

public static $a;

}

class b extends a{

public static $a; //没有这句的话，子类也使用父类的静态变量，因此$b::$a也得到12

}

$a=new a;

a::$a=12;

$b=new b;

var\_dump($a::$a); //12

var\_dump($b::$a); //12

self强制使用定义时的当前类（忽视继承），static可以根据上下文判断。

<?php

class t{

public static $\_instance=null;

final public static function getInstance(){

// if(null !== static::$\_instance){

// return static::$\_instance;

// } #注释后$b才得到tt的对象

static::$\_instance = new self(); #使用new self得到的永远是t，self强制代表当前类

return static::$\_instance;

}

}

class tt extends t{

}

$a=t::getInstance();

var\_dump($a); //得到的都是t

$b=tt::getInstance();

var\_dump($b); //得到的都是t

### 抽象类

抽象类不能实例化（任何类都可以定义为抽象类，不一定必须包含抽象方法。）

包含抽象方法的类必须定义为抽象类

抽象方法指的是只定义了其调用方式（包括参数），不能定义其具体的实现

继承一个抽象类的时候子类定义父类中的所有抽象方法；而且这些方法的访问控制符必须至少比父类更加的宽松

<?php

abstract class ab{

public $a;

public $b;

public function getA(){

return $this->a;

}

abstract protected function setA($a);

}

class a extends ab{

public function setA($a){

$this->a=$a;

}

}

### 对象接口

接口（interface）是通过interface关键字定义的，其中定义的所有方法都是空的

接口中定义的所有方法必须是公有的，这是接口的特性

接口中只可以定义常量和方法，不能定义其他变量和静态变量

接口也可以继承，可以继承多个接口（这使会扩充接口）

要实现一个接口，使用implements操作符，类可以实现多个接口

<?php

interface a{

const a=12;

public function getClas();

public function echoStr($a);

}

interface b{

const b=13;

}

interface c extends a,b{

const c=14;

}

interface d{

const d=15;

}

class ca implements c,d{

public function getClas(){

return \_\_CLASS\_\_;

}

public function echoStr($var){

var\_dump($var); //没有返回值默认返回null

}

}

$ca=new ca;

var\_dump($ca::a);

var\_dump($ca::b);

var\_dump($ca::c);

var\_dump($ca::d);

var\_dump($ca->getClas());

var\_dump($ca->echoStr("output this"));

接口可以添加类型约束，类型约束不能是int、 string 等标量和traits（其他应该都可以作为类型限制）类型约束也可以放在函数中

<?php

interface a{

public function printArr(array $arr); //限制参数的类型，扩展或者实现时定义该方法都要加上参数类型

}

class ca implements a{

public function printArr(array $c){ //必须也得加上类型限制，否则被认为不兼容

var\_dump($c);

}

}

class cb implements a{

public function printArr($cc){ //不加类型限制的话就会有fatal error

var\_dump($cc);

}

}

$ca=new ca;

$arr=[1,2,3,4,'a'];

$ca->printArr($arr);

类型约束允许null值

<?php

class A{

}

function a(A $a=null){ //允许null默认值

var\_dump($a);

}

$var=12;

$var=new A;

a($var);

### Traits

自PHP5.4开始PHP增加了Traits,以实现代码复用

Trait和类相似，但仅仅旨在用细粒度和一致的方式来组合功能，相当于是功能的集合

就相当于将类中的方法放到了外面

trait中可以使用static self 和parent关键字，但不能使用$this

<?php

trait func\_set{

private function aa($a){ //可以定义私有方法

var\_dump($a);

}

static function b(){ //可以定静态方法

//$this->aa(); //不能使用$this

}

}

class test{

use func\_set;

}

$a=new test();

$a::b(); //ok

$a->b(); //ok

var\_dump($a);

// $a->aa("trait");

从基类继承的成员被trait插入的成员所覆盖，优先级是：当前类的成员>trait的成员>被继承的方法

使得不用继承也可以拥有相同的方法，下面的两个类ca和cb拥有相同的方法组合a

<?php

trait a{

public function a(){

}

public function b(){

}

}

class ca{

use a;

}

class cb{

use a;

}

可以同时插入多个trait，之间用逗号分隔

多个trait的引入会导致方法相同的冲突错误：

如果两个trait都插入了一个同名的方法，没有解决的话会产生一个致命的错误。

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入方法中的哪一个（这使另一个方法被排除）；为了避免排除掉其他方法，也可以使用as操作符将其中的一个冲突方法一另一个名称来引入。

<?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 { //加花括号来处理trait之间的冲突

B :: smallTalk insteadof A ; //使用B的而不是A的

A :: bigTalk insteadof B ; //使用A的而不是B的

}

}

class Aliased\_Talker {

use A , B {

B :: smallTalk insteadof A ;

A :: bigTalk insteadof B ; //即使后面有将B的方法as也要此代码

B :: bigTalk as talk ;

}

}

as还可以调整方法的访问控制

除了类可以使用trait，其他的trait也可以使用trait

<?php

trait a{

public function aa(){

echo "a\aa";

}

}

trait b{ //中定义的方法会覆盖从其他trait中use的同名方法

use a;

public function aa(){

echo "b\aa";

}

}

class ca{

use b;

}

$ca=new ca;

$ca->aa(); //输出 b\aa

trait也可以定义属性，但是与方法不同的是类中不能定义同名属性。

<?php

trait A{

public $a;

}

class ca{

use A;

public $a; //定义同名属性会报错

}

trait不支持继承，可以在定义新的trait的时候use旧的trait

### 重载

PHP所提供的重载（与其他语言非常的不同）是指动态地创建类属性和方法，它是通过魔术方法来实现的。

当调用当前环境下未定义或不可见的类属性或方法时，相应的重载方法会被调用。

这些魔术方法的参数都不能通过引用传递。

属性重载魔术方法：

public void \_\_set(string $name,mixed $value)

当给不可访问的属性赋值时，\_\_set会被调用

<?php

class test{

public $a;

public function \_\_set($name,$value){

var\_dump("name: " . $name);

var\_dump("value: " . $value);

}

}

$test=new test;

$test->b=12; //不存在的属性赋值时自动调用\_\_set方法，而不会报错

public mixed \_\_get(string $name)

读取不可访问的属性的值时，\_\_get会被调用：

<?php

class test{

public $a;

public function \_\_get($name){

var\_dump("name: " . $name);

}

}

$test=new test;

var\_dump($test->b); //不可访问的属性，自动调用函数\_\_get并将结果放回

public bool \_\_Isset(string $name)

当对不可访问的属性调用isset或者empty时，\_\_isset会被调用，当在外部使用isset函数的时候，\_\_isset函数返回的结果将传递给isset函数

而empty虽然也会调用\_\_isset方法但是外部表达式的结果总是true

<?php

class test{

public $a;

public function \_\_isset($name){

var\_dump("检查不可访问的变量: " . $name);

return true;

}

}

$test=new test;

if(isset($test->b)){ //isset($test->b)的结果将有\_\_isset方法所决定

echo "\_\_isset返回了true";

}else{

echo "没有定义\_\_isset方法或者\_\_isset返回了false";

}

<?php

class test{

public $a;

public function \_\_isset($name){

var\_dump("检查不可访问的变量: " . $name);

return true;

}

}

$test=new test;

// if(isset($test->b)){

if(!empty($test->b)){ //这里的empty表达式将总是返回true，不论\_\_isset返回了什么

echo "\_\_isset返回了true";

}else{

echo "没有定义\_\_isset方法或者\_\_isset返回了false";

}

public void \_\_unset(string $name)

当对不可访问的属性调用unset时，\_\_unset方法会被调用

<?php

class test{

public $a;

public function \_\_unset($name){

var\_dump("variable to be unseted is " . $name);

}

}

$test=new test;

unset($test->b); //即使没有定义b也没有错误产生,定义了\_\_unset方法则先执行该方法

属性重载只能在对象中进行，在静态方法中这些魔术方法不会被调用，因此这些方法都不能声明为static

方法的重载：

public mixed \_\_call(string $name,array $arguments)

在对象中访问一个不可访问的方法的时候，\_\_call会被调用

<?php

class test{

public function aa(){

}

public function \_\_call($name,$args){

var\_dump("this method named " . $name . ' is called');

var\_dump("the args are: ");

var\_dump($args);

}

}

$test=new test;

$test->cc(1,2,3);

public static mixed \_\_callStatic(string $name,array $arguments)

用静态方式（单指::，不包括->尽管这也能访问静态方法）

<?php

class test{

public function aa(){

}

public static function \_\_callStatic($name,$args){

var\_dump("the static method named " . $name .' is called');

var\_dump("the args are:");

var\_dump($args);

}

public function \_\_call($name,$args){

var\_dump("the method named " . $name .' is called');

var\_dump("the args are:");

var\_dump($args);

}

}

$test=new test;

// $test::bb(1,2,3,4); //只有这样才会认为是调用静态方法

$test->bb(1,2,3,4,5); //这只会认为是调用普通方法

property\_exists可以判断类的属性是否存在

### 遍历对象

可以使用foreach语句遍历所有可见（在类外面：可见只包含public的，不包括static尽管是public的；在类内部：可见包含所有）的属性

<?php

class test{

public $a;

public $b;

protected $c;

private $d;

public static $f;

public function aa(){

}

}

$test=new test;

var\_dump($test::$f); //随然能够访问但是并不能遍历出来

foreach($test as $k => $v){

var\_dump("k: " . $k . " v: " . $v); //只能遍历public且不是static的

}

<?php

class test{

public $a;

public $b;

protected $c;

public static $d;

private $f;

public function getP(){

foreach($this as $k => $v){

var\_dump("k: " . $k . ' v: ' . $v); //仍然不能遍历静态属性

}

}

}

$test=new test;

foreach($test as $k => $v){

var\_dump("k: " . $k . ' v: ' . $v);

}

echo "类内部" . PHP\_EOL;

$test->getP();

可以实现Iterator接口来决定如何遍历以及每次遍历时哪些值可用：

也可以用IteratorAggregate接口替代实现所有的Iterator方法

更多参看SPL迭代器扩展

### 魔术方法

\_\_sleep

Public array \_\_sleep(void)

Serialize()函数会检查类中是否存在一个魔术方法\_\_sleep,如果存在则先执行该方法然后之心序列化操作，此功能可以用于清理对象并返回一个包含对象对象所有应被徐丽华的变量名称的数组。

\_\_sleep不能返回父类的私有成员的名字，可以用serializable替代

<?php

class test{

public $a;

public $b;

public static $c;

public function d(){

}

public function \_\_sleep(){

var\_dump("执行序列化");

return ['a'];

}

}

$test=new test;

var\_dump(serialize($test));

\_\_wakeup

Public void \_\_wakeup(void)

Unserialize会检查是否存在一个\_\_wakeup方法，如果存在则先调用\_\_wakeup方法，预先准备对象需要的资源。

<?php

class test{

public $a;

public $b;

private $c;

public static $d;

const f=12;

public function e(){

}

public function g(){

var\_dump(\_\_CLASS\_\_ . '/' . \_\_FUNCTION\_\_);

}

public function \_\_wakeup(){

var\_dump("unserialize");

}

}

$test=new test;

var\_dump($test);

$cc=serialize($test);

var\_dump($cc);

$dd=unserialize($cc);

var\_dump($dd);

$dd->g();

\_\_toString

Public string \_\_toString(void)

该方法定义当类被当做一个字符串时应该执行的程序

该方法必须返回一个字符串

<?php

class test{

public function \_\_toString(){

return \_\_CLASS\_\_ . '的对象'; //返回到echo

}

}

$test=new test;

echo $test;

\_\_invoke

Mixed \_\_invoke([$...])

当尝试以调用函数的方法调用一个对象时，\_\_invovke方法回被自动调用

参数会作为数组传递到\_\_invoke方法

<?php

class test{

public function \_\_invoke($a,$b){ //一个一个参数匹配

var\_dump($a);

var\_dump($b);

}

}

$test=new test;

$test("12","aa");

var\_dump(is\_callable($test)); //true，未定义invoke方法则返回false

\_\_set\_state

Static object \_\_set\_state(array $properties)

当调用var\_export导出类时，此静态方法会被调用

该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

class test{

public $a;

public $b;

public static function \_\_set\_state($arry){

$t=new test;

var\_dump($arry);

$t->a=$arry['a'];

$t->b=$arry['b'];

return $t;

}

}

$tt=new test;

$tt->a=12;

$tt->b="cc";

eval('$b=' . var\_export($tt,true) . ';');

var\_dump($b);

### final关键字

如果父类中的方法被声明为final,则子类不能覆盖该方法；如果一个类被声明为final，则不能被继承。属性不能定义为final，只有类和方法才能被定义为final

<?php

class base{

final public function a(){

}

}

final class child extends base{

public function a(){ //会报错

}

}

class cc extends child{ //不能继承final类

}

### 对象复制

对象复制可以通过关键字clone来完成（如果可能它将调用对象中的\_\_clone方法）

对象中的\_\_clone方法不能被直接调用

当对象被复制后，PHP5会对对象的所有属性执行一个浅复制，所有的引用属性仍然会是一个指向原来变量的引用。

Void \_\_clone(void)

当复制完成时，如果定义了\_\_clone方法则创建的对象（复制生成的对象）中\_\_clone方法会被调用，可用于修改属性的值

<?php

class test{

public $a;

public function \_\_construct($a){

$this->a=$a;

}

public function \_\_clone(){

var\_dump("clone one object");

}

}

$test=new test("12");

$test2=clone($test);

$test2->a="aa";

var\_dump($test);

var\_dump($test2);

### 对象的比较

当==比较两个对象变量时，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，那么这两个对象相等；如果使用全等===比较，这两个对象变量要指向某个类的同一个实例（即同一个对象）

在扩展中也可自行定义。

### 类型约束

PHP5可以使用类型约束，函数的参数可以指定必须为对象、接口、数组或者callable；如果使用null作为参数的默认值，那么在调用时仍然可以传递null作为实参

类型约束不能用于标量类型如int string，traits也不可以。

<?php

class a{

}

function test(a $a,array $arr,callable $func){

var\_dump("函数参数类型约束");

}

$a=new a;

$arr=[];

function t(){

}

test($a,$arr,'t'); #函数类型参数要传递函数名

类型约束可以使用null值，但要在定义时指定默认值null才可以在调用时传入null

### 后期静态绑定

Static::不再被解析为定义当前方法所在的类，而是实际运行时计算的。

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，也即当静态调用使用parent::或者self::时将转发到定义他们的方法调用。

<?php

class a{

public static function who(){

var\_dump(\_\_CLASS\_\_);

}

public static function test(){

// self::who();

static::who();

}

}

class b extends a{

public static function who(){

var\_dump(\_\_CLASS\_\_);

}

}

b::test(); //输出a，这就是self的效果;当将self换为static时则输出b

static::只能用于静态属性，但可以用于非静态和静态方法

<?php

class a{

function test(){

var\_dump(\_\_FUNCTION\_\_);

}

public function test2(){

static::test();

}

}

$a=new a;

$a->test2();

### 对象和引用

默认情况下对象都是通过引用传递的

<?php

class a{

public $a;

public $b;

}

function test(a $a){

$a->a="aa";

$a->b="bb";

}

$a=new a;

$a->a='a';

$a->b='b';

var\_dump($a); a b

test($a);

var\_dump($a); aa bb

对象变量保存一个标识符指向对象的真正内容，因此当复制时仅仅是标识符的拷贝，引用的对象还是一个。

### 对象序列化

所有PHP里面的值都可以使用函数serialize来返回一个包含字节流的字符串表示。

Unserialize可以将字符串变回PHP原来的值。序列化一个对象将保存对象的所有变量，但不会保存对象的方法，只会保存类的名字。

Unserialize一个字符串时，这个字符串代表的对象的类必须被定义。

<?php

class a{

public $a;

public $b;

public static $c;

const D=12;

public function aa(){

}

public static function bb(){

}

}

$a=new a;

$a->a=12;

$a->b=13;

$a::$c=1;

var\_dump($a);

$sa=serialize($a);

var\_dump($sa);

$ua=unserialize($sa);

var\_dump($ua);

var\_dump($ua::$c); //输出1

#### 命名空间

PHP在5.3.0以后的版本开始支持命名空间。如果没有定义命名空间，则默认所定义的类和方法属于全局空间(\)

<?php

namespace a\b\c;

class a{

public function \_\_construct(){

var\_dump(\_\_CLASS\_\_);

var\_dump(\_\_FUNCTION\_\_);

}

}

function aa(){

var\_dump(\_\_FUNCTION\_\_);

}

new a; #string(7) "a\b\c\a" string(11) "\_\_construct"

aa(); #string(8) "a\b\c\aa"

命名空间就是一种封装事务的方法（如文件系统的分目录规则，一个目录就是一个命名空间，可以包含相同的文件名，这个原理用到程序设计领域就是命名空间的概念）。

PHP命名空间提供了一种将相关的类、函数和常量组合在一起的途径。

<?php

namespace my\name;

class a{

}

function func(){

}

const MYCONST=12;

$a=new \my\name\a;

var\_dump($a);

var\_dump(namespace\MYCONST);

在文件定义的命名空间的根空间时属于全局空间的：

<?php

namespace foo;

function bar(){

echo \_\_NAMESPACE\_\_ . '\\' . "methods: " . \_\_FUNCTION\_\_;

}

\foo\bar(); //不能使用foo\bar,会将foo\bar附属到当前的命名空间下，即foo\foo\bar

//这个foo是属于全局命名空间的，在开始加上\

使用命名空间调用函数或类的时候，除非是想要调用当前命名空间的子空间，否则需要在开始加上代表全局空间的符号：\

尽管任何合法的PHP代码都可以放在命名空间中，但是只有类、函数和常量收到命名空间的影响。

命名空间需要在文件头声明，可以声明咋命名空间之前的唯一合法代码是定义源文件编码方式的declare语句。

同一命名空间可以出现在在多个文件中，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。

<?php

declare (encoding='UTF-8'); #当配置文件开启了multibyte时可以定义此语句

namespace n;

class a{

}

function aa(){}

const ACONST=12;

命名空间中不能用关键词如：function abstract等

define定义的常量属于全局命名空间，无论定义常量时代码所在的命名空间时什么。而const定义的常量属于所在命名空间的常量：

<?php

namespace test;

class a{

}

define("ACONST","CONST VARIABLE"); #这样定义的常量是属于全局的

const ACONST=12; #这样定义的常量是属于命名空间的，这样就定义了本命名空间的常量

if(defined("ACONST")){

var\_dump(ACONST); //输出12，因为默认本地命名空间

$n='\ACONST';

echo constant($n); //输出CONST VARIABLE，因为是全局命名空间

}else{

var\_dump("not defined");

}

const函数返回一个常量的值，类内部定义或者外部定义的：

[mixed](http://php.net/manual/en/language.pseudo-types.php#language.types.mixed) **constant** ( string $name )

### 定义子命名空间

与目录和文件的关系很像，命名空间的名字也可以使用分层次的方式定义：

<?php

namespace myproject\sub\_pro\a\b; #定义一个子命名空间

class a{}

function aa(){}

const THECONST=12;

$constName=\_\_NAMESPACE\_\_ . '\THECONST';

var\_dump(constant($constName)); //输出12

### 在同一个文件中定义多个命名空间

可以在同一个文件中定义多个命名空间，有两种语法，建议使用第二种语法即在命名空间名字后面加上花括号{}

<?php

namespace pa;

class a{};

function aa(){}

const MYCONST="CLASS\_pa";

namespace pb; //之后书写的多有代码如果没有新的命名空间定义，将都属于pb

class a{}

function aa(){}

const MYCONST="CLASS\_pb";

var\_dump(MYCONST); //输出pb

var\_dump(\pa\MYCONST); //pa

var\_dump(\pb\MYCONST); //pb

建议使用第二种方法来在同一个文件中定义命名空间，且命名空间的花括号不能有任何PHP代码，因此需要将代码都定义在命名空间中，当然 可以使用不带名称的namespace代表全局空间：

<?php

namespace pa{

class a{};

function aa(){}

const MYCONST="CLASS\_pa";

}

namespace pb{

class a{}

function aa(){}

const MYCONST="CLASS\_pb";

}

namespace{

var\_dump(MYCONST); //输出pb,这样定义后所有的代码都必须在命名空间的花括号中,无名称的命名空间代表全局命名空间

var\_dump(\pa\MYCONST); //pa

var\_dump(\pb\MYCONST); //pb

}

### 使用命名空间

类似文件系统中文件的引入，命名空间的引入有三种方式：

1. 非限定名称，或不包含前缀的类名称

将被解析为当前命名空间，如果当前命名空间不存在函数或常量的定义则进一步被解析为全局函数或常量名称。

<?php

namespace name192;

const FOO='name192\_12';

define("FOO","12");

var\_dump(FOO); #这会现寻找当前命名空间的常量，没有找到则寻找全局同名常量

echo constant("FOO"); #这样获取常量值的方式必须带有命名空间，否则被认为是全局

2、限定名称或包含前缀的名称（类似当前目录的同级目录的下级目录）

<?php

namespace mypro;

require\_once './192.php';

// use sub\_pro\name192;

class a{}

$a=new sub\_pro\name192\a;

var\_dump($a);

echo FOO;

3、完全限定名称（类似绝对路径名）

<?php

namespace mypro;

class a{

}

$a=new \stdClass; #前反斜杠表示使用的全局命名空间

var\_dump(\strlen("afadfaf")); //使用全局的函数

如果当前方法找不到也会寻找全局同名方法：

<?php

namespace mypro;

class a{

}

$a=new \stdClass; #前反斜杠表示使用的全局命名空间

function strlen(){ //将strlen方法重新定义了

var\_dump("mypro\strlen");

}

var\_dump(strlen());

var\_dump(\strlen("afadfaf")); //完全限定名称访问的仍然是全局的strlen方法

当使用非限定名称及不包含前缀时会首先在当前命名空间查找，没有的话再到全局命名空间查找，这只适用于常量和方法，类的话只会在当前命名空间查找：

<?php

namespace mypro;

class a{

}

$a=new \stdClass; #前反斜杠表示使用的全局命名空间

function strlen(){ //将strlen方法重新定义了

var\_dump("mypro\strlen");

}

var\_dump(strlen());

var\_dump(\strlen("afadfaf")); //完全限定名称访问的仍然是全局的strlen方法

访问任意全局类、函数或常量都可以使用完全限定名称（即带全局命名空间的格式）

<?php

namespace MyProject {

class Person {}

}

namespace MyProject\People {

class Adult extends ..\Person {} #这样写是错误的

}

?>

命名空间只对三个类型起作用：类-class 、函数-function 和常量

至于其他的并不受命名空间影响，就像在全局空间定义使用一样。

<?php

namespace main{

}

namespace main\sub1{

$data=1;

}

namespace main\sub2{

echo $data; // 1

$data=2;

}

namespace main\sub1{

echo $data; //2

$data=1;

}

namespace {

echo $data; //1

}

### 命名空间和动态语言的特征

在动态访问类、函数或者常量时，必须在名称前加上限定前缀，全局反斜杠可要可不要。

在用字符串动态访问类、函数或者常量的时候要加上限定名。否则就会到全局查找：

<?php

namespace {

const constName="global";

}

namespace t{

class cls{

function \_\_construct(){

var\_dump(\_\_METHOD\_\_);

}

}

function funcname(){

var\_dump(\_\_FUNCTION\_\_);

}

const constName="local";

$a="t\cls"; #可以不用加全局反斜杠，\t\cls

$cls=new $a;

$fname="t\\funcname";

$fname();

echo constant('constName'); #访问全局

echo constant('t\constName'); #访问local

}

### namespace关键字和\_\_NAMESPACE\_\_常量

\_\_NAMESPACE\_\_魔术常量的值包含当前命名空间的名称。当在全局时，不包括任何命名空间中的代码，它包含一个空的字符串。可以使用这个魔术常量来判断当前是否在全局空间。

<?php

namespace{

var\_dump(\_\_NAMESPACE\_\_); //输出 ""

}

namespace cc{

var\_dump(\_\_NAMESPACE\_\_); //输出cc

}

\_\_NAMESPACE\_\_多用来创建类名（带空间）等。

namespace关键字可以用来显示访问当前命名空间或子空间中的元素，就像类中的self

<?php

namespace cc\aa{

var\_dump(\_\_NAMESPACE\_\_); //输出 ""

function ac(){

var\_dump("namespace:aa");

}

}

namespace cc{

use aa;

var\_dump(\_\_NAMESPACE\_\_); //输出cc

function ac(){

var\_dump("namespace:cc");

}

namespace\aa\ac();

}

### 使用命名空间：别名/导入

允许通过别名引用或导入外部的完全限定名称

有两种使用别名或导入方式：为类名使用别名；为命名空间使用别名。即使用use可以导入类和命名空间。

PHP不支持直接导入函数或常量。

别名关键字 use

<?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 ;

$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

?>

PHP还支持在一行使用多个use语句

<?php

namespace abc;

use a\b\c as cc,a\b;

导入操作是在编译的时候执行的，但动态类的名称、函数名或者常量则不是。

动态的导入的总是全局的，除非明名称字符串包含命名空间。

导入类：

<?php

namespace a;

class test{

}

namespace b;

use \a\test; //将命名空间a的类test导入进来了

$t=new test; //实例化了a的test

var\_dump($t);

不能在方法或者函数中使用use关键字

如果没有定义任何命名空间，所有的类和函数的定义都是在全局空间。在名称之前加上前缀\表示使用全局空间的名称。

<?php

namespace a{

function cc(){

var\_dump("local");

}

namespace\cc();

\cc(); #使用全局函数cc

}

namespace{

function cc(){

var\_dump("global");

}

}

在一个命名空间中，当PHP遇到一个非限定类（就是没有带命名口空间）的类 函数 或者常量时，它使用一种有限策略来访问相应的变量：

1、类名称总是解析到当前命名空间中的名称，如果不存在则报错；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。

<?php

namespace a;

$a=new stdClass; #这个类总是被解析为当前命名空间

$aa=new \stdClass; #这样才能使用到全局空间的标准类

1. 对于函数或者常量来说，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。

<?php

namespace{

const ACONST="GLOBAL\_12";

}

namespace a{

// const ACONST=12;

var\_dump(ACONST); //本地没有则访问全局

}

1. 除了类、函数和常量之外，其他都不受命名空间的影响，就像是定义和访问全局空间。

### 名称解析规则

## 异常处理

try …catch…catch…finally

Exception类：

<?php

$e=new Exception;

var\_dump($e);

输出：

class Exception#1 (7) {

protected $message =>

string(0) ""

private $string =>

string(0) ""

protected $code =>

int(0)

protected $file =>

string(59) "E:\deleting\enhancingProgram\PHP\codes\yuyan\_cankao\205.php"

protected $line =>

int(2)

private $trace =>

array(0) {

}

private $previous =>

NULL

}

## 生成器

生成器允许你在 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, 那会使你的内存达到上限，或者会占据可观的处理时间。相反，你可以写一个生成器函数，就像一个普通的自定义函数一样, 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，以便生成需要迭代的值。

将range实现Wie生成器xrange

<?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  " ;  
}  
?>

## 引用的解释

在PHP中引用意味着用不同的名字访问同一个变量的内容

在PHP中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

这也说明变量名和变量的内容是存储在不同的地方的

PHP中的引用允许两个变量来指向同一个内容

<?php

$a=12;

$b=&$a; //这时a和b指向了同一个地方

var\_dump($a);

var\_dump($b);

$a="12";

var\_dump($b);

如果具有引用数组被拷贝，其值不会解除引用，当数组传值给函数也是如此

<?php

$arr=[1,2,3];

$a=12;

$arr['arr']=&$a;

var\_dump($arr);

$arr2=$arr; //复制的时候引用也被复制了

var\_dump($arr2);

$a="12";

var\_dump($arr2); //$arr2['arr']与$a指向了同一个地方

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。

<?php  
function  foo (& $var ) { }  
  
foo ( $a );  // $a is "created" and assigned to null

New关键字自动返回引用，因此变量指向的是对象

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；而非引用变量则不会改变，要想改变需要使用for语句。

<?php

$arr=[1,2,3,4];

foreach($arr as $k => $v){

$v \*=2;

}

var\_dump($arr); //还是1 2 3 4，非引用不改变

foreach($arr as $k => &$v){

$v \*=2;

}

var\_dump($arr); //改变了

另一个需要引用的就是引用传递变量

<?php

function foo(&$var){ #定义的时候声明为引用即可

$var++;

}

$a=12;

foo($a);

var\_dump($a); //13

可以通过引用传递的有：

变量 、new语句如foo(new foobar())

除此之外的任何表达式都不能通过引用传递

<?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  
?>

引用不是指针

### 引用返回

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(){ //返回引用&

return $this->value;

}

}

$foo=new foo;

$myValue=&$foo->getValue(); //调用的时候也要加，表示为myValue赋值了一个引用

$foo->value=12;

echo $myValue; //输出12

注意使用return ($this->value)不能达到引用的效果，因为这是返回一个表达式达的结果而不是一个引用变量。

取消引用

使用unset

当unset一个引用，只是断开了变量名和变量内容之间的绑定，但这并不意味着变量内容被销毁了。

PHP中其他使用引用的结构有：global引用，在函数内部声明global时实际建立了一个 到全局变量的引用，等价于：

$var=&$GLOBALS[‘var’]

所以unset($var)不会unset全局变量

$this也是对调用它的对象的引用。

<?php

class a{

public $a=12;

function \_\_construct(){

var\_dump($this->a);

$that=$this;

unset($this); //取消了$this到对象的引用

var\_dump($that);

}

}

new a;

## 预定义变量

### 超全局变量

超全局变量是在全部作用域中始终可用的内置变量，在函数内部也可用，包括如下：

$GLOBALS

引用全部作用域中可用的全部变量（包括下面的超全局变量）

$GLOBALS包含PHP中的所有预定义变量和在全局作用域定义的变量，变量名就是数组的键。

<?php

var\_dump($GLOBALS); //$GLOBALS超全局变量包含所有的预定义变量和全局变量

$a=12;

var\_dump($GLOBALS['a']); //输出12

$\_SERVER

包含服务器和执行环境信息，这个变量中包含了诸如头信息、路径以及脚本位置等等信息。

这个数组中的项有web服务器创建。

<?php

var\_dump($GLOBALS['\_SERVER']);

var\_dump($\_SERVER);

$\_GET

通过url参数传递给当前脚本的变量的数组。

没有参数传递时这是一个空数组

$\_GET是通过urldecode传递的，即它自动将传来的参数执行urldecode

$\_POST

通过http post方法传递给当前脚本的变量的数组

没有参数传递时这是一个空数组

<?php

var\_dump($\_POST);

$\_FILES

通过http post方式上传到当前脚本的项目的数组（上传文件时）

$\_REQUEST

默认情况下包含了$\_GET $\_POST和$\_COOKIE的数组(仅仅包含这些变量从服务器传来的变量或值，如果是在脚本内部增加的值就不会再$\_REQUEST里面体现)

<?php

$\_GET['a']="aa";

var\_dump($\_GET);

var\_dump($\_REQUEST);

$\_COOKIE

通过http cookie方式传递给当前脚本的变量的数组

<?php

var\_dump($\_COOKIE);

$php\_errormsg

该变量包含有PHP生成的最细错误消息，这个变量只有在错误发生的作用域内才是可用的，并且要求track\_errors配置项是开启的。

<?php

@strpos();

echo $php\_errormsg ; //输出: strpos() expects at least 2 parameters, 0 given

$\_SESSION

当前脚本用session变量的数组

$\_ENV

通过环境方式传递给当前脚本的变量的数组

<?php

var\_dump($\_ENV);

$HTTP\_RAW\_POST\_DATA

原生post数据，该变量包含post提交的原始数据

<?php

$str=($HTTP\_RAW\_POST\_DATA);

file\_put\_contents('2.txt',$str);

echo 'ok';

$http\_response\_header

<?php

file\_get\_contents('http://www.baidu.com');

var\_dump($http\_response\_header); //此时此变量包含的是baidu网响应的头部

类似于get\_headers():

array **get\_headers** ( string $url [, int $format = 0 ] )

取得服务器响应一个http请求所发送的标头（以索引数组或关联数组的形式），如果失败返回false，可选参数可以选择值为0或者1

<?php

var\_dump(get\_headers('http://www.baidu.com',0)); //索引数组

var\_dump(get\_headers('http://www.baidu.com',1)); //关联数组

$argc

包含当前运行于命令行下时传递给当前脚本的参数的数目

注意：脚本的名称总是作为参数传递给脚本，因此$argc的最小值为1

这个变量在仅在register\_argc\_argv打开时可用

<?php

var\_dump($argc); //1 第一个默认为当前脚本的名称

$argv

传递给脚本的参数数组

包含当运行于命令行下时传递给当前脚本的参数的数组

注意：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

这个变量仅在register\_argc\_argv打开时可用

<?php

var\_dump($argv);

输出：

array(4) {

[0] =>

string(7) "227.php"

[1] =>

string(3) "a=1"

[2] =>

string(3) "b=1"

[3] =>

string(3) "c=1"

}

getopt — 从命令行参数列表中获取选项（也可以将选项当做传入的参数）

## 预定义异常

### Exception

exception是所有异常的基类

类摘要：

**Exception** {

/\* 属性 \*/

protected string [*$*message](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html#exception.props.message);

protected int [*$*code](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html#exception.props.code);

protected string [*$*file](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html#exception.props.file);

protected int [*$*line](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html#exception.props.line);

/\* 方法 \*/

public [\_\_construct](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.construct.html) ([ string $message = "" [, int $code = 0 [, [Exception](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html) $previous = **NULL** ]]] )

final public string [getMessage](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getmessage.html) ( void ) ##获取异常消息内容

final public Exception [getPrevious](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getprevious.html) ( void ) ##获取异常链中的前一个异常

final public int [getCode](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getcode.html) ( void ) ##获取异常代码

final public string [getFile](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getfile.html) ( void ) ##获取发生异常的程序文件名称

final public int [getLine](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getline.html) ( void ) ##获取发生异常的代码在文件中的行号

final public array [getTrace](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.gettrace.html) ( void ) ##获取异常追踪信息

final public string [getTraceAsString](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.gettraceasstring.html) ( void ) ##获取自符串类型的异常追踪信息

public string [\_\_toString](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.tostring.html) ( void ) ##将异常对象转换为字符串

final private void [\_\_clone](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.clone.html) ( void ) ##异常克隆

}

<?php

$ex=new Exception;

var\_dump($ex);

输出：下面仅仅输出了属性

class Exception#1 (7) {

protected $message =>

string(0) ""

private $string =>

string(0) ""

protected $code =>

int(0)

protected $file =>

string(59) "E:\deleting\enhancingProgram\PHP\codes\yuyan\_cankao\228.php"

protected $line =>

int(2)

private $trace =>

array(0) {

}

private $previous =>

NULL

}

### ErrorException

错误异常，该异常继承自Exception

**ErrorException** extends [Exception](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html) {

/\* 属性 \*/

protected int [*$*severity](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.errorexception.html#errorexception.props.severity); ##异常级别，这个是相对于Exception而言自定义的属性

/\* 方法 \*/

public [\_\_construct](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/errorexception.construct.html) ([ string $message = "" [, int $code = 0 [, int $severity = 1 [, string $filename = \_\_FILE\_\_ [, int $lineno = \_\_LINE\_\_ [, [Exception](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/class.exception.html) $previous = **NULL** ]]]]]] )

final public int [getSeverity](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/errorexception.getseverity.html) ( void ) ##获取异常的严重程度

/\* 继承的方法 \*/

final public string [Exception::getMessage](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getmessage.html) ( void )

final public Exception [Exception::getPrevious](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getprevious.html) ( void )

final public int [Exception::getCode](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getcode.html) ( void )

final public string [Exception::getFile](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getfile.html) ( void )

final public int [Exception::getLine](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.getline.html) ( void )

final public array [Exception::getTrace](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.gettrace.html) ( void )

final public string [Exception::getTraceAsString](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.gettraceasstring.html) ( void )

public string [Exception::\_\_toString](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.tostring.html) ( void )

final private void [Exception::\_\_clone](mk:@MSITStore:E:\deleting\enhancingProgram\PHP\php_manual_zh_review.chm::/res/exception.clone.html) ( void )

}

<?php

$err\_ex=new ErrorException;

var\_dump($err\_ex);

输出：仅仅输出了属性

class ErrorException#1 (8) {

protected $message =>

string(0) ""

private $string =>

string(0) ""

protected $code =>

int(0)

protected $file =>

string(59) "E:\deleting\enhancingProgram\PHP\codes\yuyan\_cankao\229.php"

protected $line =>

int(2)

private $trace =>

array(0) {

}

private $previous =>

NULL

protected $severity =>

int(1)

}

## 预定义接口

### 遍历接口

检测一个类是否可以使用foreach进行遍历的接口；该接口是抽象接口，无法被单独实现，必须实现有IteratorAggegate或者Iterator接口才可以

这个接口没有任何方法，仅仅是作为所有可遍历类的基本接口

实现此接口类可以是哟呵那个foreach进行遍历（当然实际需要实现**IteratorAggregate** 或 **Iterator** 接口）

### 上下文（Context）选项和参数

### 支持的协议和封装协议